Performance Analysis of Real-Time Task Systems using Timed Automata

Georgeta Igna

Typeset with LaTeX 2ε
Printing: Ipskamp Drukkers
Cover design : Michiel Stomphorst, Ideaal Communicatie

This research has been carried out as part of the Octopus project with Océ Technologies B.V. under the responsibility of the Embedded Systems Institute. This project was partially supported by the Netherlands Ministry of Economic Affairs under the Bsik program.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

IPA dissertation series 2013-02

Performance Analysis of Real-Time Task Systems using Timed Automata

Een wetenschappelijke proeve op het gebied
van de Natuurwetenschappen, Wiskunde en Informatica.

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. mr. S.C.J.J. Kortmann,
volgens besluit van het college van decanen
in het openbaar te verdedigen op dinsdag 22 januari 2013
om 10.30 uur precies
door

Georgeta Igna

geboren op 15 februari 1983
te Arad, Roemenië

Promotor:

Prof. dr. F.W. Vaandrager

Manuscriptcommissie:

Prof. dr. E. Ábráham	RWTH Aachen University
Prof. dr. ir. T. Basten	TU Einhoven & Embedded Systems Institute
Prof. dr. J.J.M. Hooman	Radboud University Nijmegen & Embedded Systems Institute
Dr. A. David	Aalborg University
Dr. ing. M. Minea	Politehnica University of Timișoara

ISBN 978-94-6191-580-1

Contents

Chapter 1

Introduction

The electronic devices that we interact with on a daily basis have become ubiquitous and have infiltrated many aspects of our life. Thinking about the future, it can be expected that mobile and embedded systems will be even more computationally powerful, permitting new applications. *Embedded systems* are the software/hardware units that control systems such as consumer electronics, medical equipments, telecommunication or transportation systems. Embedded systems are tightly coupled with their environment and often have to satisfy constraints imposed by it. In addition, they have to meet particular requirements in terms of power consumption, size, and weight. One class of embedded systems are *real-time systems*, which have assigned timing constraints for their interaction with the real world. A system designed to meet strict timing requirements is often referred to as a *hard real-time system*. An example of hard real-time is an air-bag for a car. In contrast, a system for which occasional violations of timing constraints are acceptable is known as a *soft real-time system*. For example, usually a laser printer is rated by pages-per-minute, but it can take different times to print a page (depending on the "complexity" of the page) without harming the machine or the user.

Given a set of requirements, the design of a corresponding system calls for the construction of an abstract representation that defines the essentials for the final result. As pointed out in [HS07], embedded systems design is not a straightforward extension of hardware and software design. Hardware components are in general more static than software components, whose structure might change at runtime. Designing hardware and software separately increases the chances that errors remain after the design phase that may require rethinking of the design. An error detected late may cause serious impact on the credibility and revenue of the producer company and even loss of human lives in case of safety critical systems. Therefore, a good embedded systems design should assume a close interaction between hardware and software, aiming at optimizing the software to meet requirements specific to the hardware.

Further aspects that complicate the design process of embedded systems are:

- Embedded systems are usually designed in a short period of time due to the strong competition established in the market. The quicker a product appears on the market, the larger the number of items sold, which increases the profits.

- Due to the close interaction with the environment, it is not enough to find design solutions that guarantee all functional constraints of embedded systems; non-functional constraints (e.g. reliability, security, energy consumption, and reaction time) are equally important.

- Although embedded systems are known to perform the same tasks, heterogeneous systems have received great interest from industry [HSA]. Heterogeneous systems can play different roles in addition to their specialized functionality boosting the performance of a system.

- Different parts of a design might contain different level of details.

- Embedded systems must operate under unpredictable conditions. This implies that predictability and robustness are essential requirements to put into consideration during design.

1.1 Performance Analysis

One of the major challenges in embedded system design is finding the essential characteristics the final product should meet before entering into the implementation details [Thi07]. To achieve this, the engineer should decide early on aspects like: a mapping or scheduling of applications in hardware, a rough division of the functions implemented in software and hardware, or identify the resource that acts as a bottleneck.

In this study, we investigate the performance analysis of real-time printing systems using formal verification techniques. *Performance analysis* is the evaluation process of different design options in order to find optimal solutions with respect to metrics such as timing constraints, memory or energy consumption.

Automated design methods form the alternative to the traditional methods that heavily rely on earlier experience from similar products and manual designs, which cannot keep up with the complexity of today's designs. *Model-based design* is an example of such methods used for the development of embedded systems. Instead of prototypes, this approach favors the use of (system-level) models for design and in some cases can generate implementations directly from the models. This approach helps in cutting development costs and time, because various system configurations are easily evaluated once a model has been built, and models can be shared between related products or different modeling stages.

Simulation is one of the model-based techniques frequently used in industry. Simulation-based methods can represent systems in a very detailed manner. However, simulations can reproduce only a finite set of system runs, which makes simulation-based approaches impractical when proving hard system performance guarantees such as worst-case or best-case execution times.

Even though designers typically do not have the time or inclination to learn new technologies, recently we can observe a trend towards more designers using tools that implement formal methods. *Formal methods* are mathematical techniques for the specification, design and verification of software and hardware systems. Formal methods can be used in many phases of a product production: writing formal specifications, proving properties about specifications, deriving implementations from a given specification, verifying a specification with respect to a given implementation [Sch04]. *Formal verification* is the process of proving whether a system, represented as an abstract mathematical model, satisfies a given property (i.e. requirement) formally described with mathematical rigor. Verification implies an exploration of all possible system behaviors [CGP01]. An important benefit of using formal verification in the design process is getting proofs for critical parts of a system that are not easy to verify with simulation, or for modules used at different places. One of the best known verification techniques is model checking. *Model checking* is an automated method for enumeration of all (explicit or implicit) states reachable in a system. We are going to use model checking for real-time system performance evaluation. The point here is that model checking will provide not only estimates of the performance, but performance guarantees.

A plethora of formal methods techniques come to help in the performance analysis of embedded systems e.g. Colored Petri Nets, Parallel Object-Oriented Specification Language (POOSL), Synchronous Dataflow (SDF), Symbolic Timing Analysis for Systems (SymTA/S), Modular Performance Analysis (MPA), and Timed Automata. These techniques treat the same problem, namely performance analysis of embedded systems, but from a different angle.

Colored Petri Nets (CPNs) [Jen92] are a discrete event modeling language that extends Petri nets with a functional programming language called Standard ML. This extension gives the expressivity of a full (functional) programming language with support for data variables. A CPN model is a collection of modules each containing a network of places, transitions, and arcs. Arcs link places with transitions and the other way around. Places may contain a discrete number of tokens. A transition in a Petri net may fire whenever there are sufficient tokens at the start of all input arcs, then, it consumes these tokens and places tokens at the end of all output arcs. CPN have been applied in many domains like: workflow and business processes [vdAvH02], control systems [FJT07], protocols [DL08, EKK08], embedded systems [JCTX06]. CPN models can be simulated interactively or automatically with CPN Tools [JKW07].

POOSL [PV97] is a system-level description language. It is based on a timed version of process algebra and basic concepts of object-oriented programming lan-

guages. A POOSL model is a combination of process objects and data objects. Process objects can exchange messages via channels. They can also be organized in clusters. A process object may contain (local) data objects that can be passed to other process objects. The POOSL language has been successfully applied for modeling and analysis of many industrial systems, such as network processors [TVK03], printer controllers and car navigation systems [FVVC06]. SHESim [SHE, TFG+07] offers graphical specification and interactive simulation for POOSL models.

SDF [LM87] corresponds to the weighted marked graphs subclass of Petri nets [TCWCS92]. Vertices of an SDF graph are called actors. An actor consumes or produces a constant number of tokens for each firing. In a timed SDF, each actor is annotated with a number that represents its worst-case execution time. Timed SDFs can be analyzed efficiently for many performance metrics, such as maximum throughput [GGS+06], latency [GSB+07], minimum buffer sizes [SGB08] or trade-offs between performance and resource requirements [YGB+10]. One of the tools that allows to perform analysis on SDF graphs is SDF3 [SGB06].

SymTA/S [RJE03] uses real-time scheduling analysis techniques for the local analysis of event models and event model interfaces or event adaption functions for inter-component analysis. SymTA/S offers analysis for standard event models and more recently for more general event models [SRIE08]. Standard event models are defined using three parameters: period between events, jitter for the situations when events have some variations around the period, and distance between two consecutive events, in case the jitter is larger than the period. SymTA/S, as part of the commercial tool called Symtavision, has been used for timing and scheduling analyses in the automotive and avionics industries [Sym].

MPA [CKT03] is a framework for worst-case performance evaluation of distributed embedded systems [SBYT12, PLT11, STSB10, LPT09]. An MPA model comprises basic building blocks that input a set of lower and upper arrival curves (representing the number of events that occur in a given time interval)and a set of lower and upper service curves for each resource (representing the resource capacity available in the same time interval) for which it provides corresponding output curves using Real-Time Calculus (RTC) [TCN00]. RTC is an extension of Network Calculus [BT01] which uses min-plus and max-plus calculus to obtain output arrival and service curves.

SymTA/S and MPA allow compositional reasoning. This approach applies to systems that can be decomposed into multiple (concurrent) processes. A system property can be divided into sub-properties regarding small parts of the system such that an overall conclusion is deduced from the composition of local analysis results.

Timed automata [AD94] are finite-state machines extended with real-valued variables called clocks. Uppaal [BDL04] is a timed automata model checker, whose language extends timed automata with additional features such as bounded integer variables, synchronization channels and user-defined functions. The tool offers a graphical interface for designing models, a simulator and a verification module

with the possibility of visualizing counter-examples. UPPAAL [BDL04] has been successfully applied in many domains, e.g. optimal scheduling [AKM03, MLR+10], performance analysis of real-time distributed systems [HV06, PWT+07], protocol verification [BGVZ11] and controller synthesis [CJL+09].

Perathoner et al. [PWT+07] present a comparison between MPA, SymTA/S and UPPAAL based on a set of benchmarks for distributed real-time systems. The results show that UPPAAL gives exact performance predictions. The compositional methods provide pessimistic results due to their limited modeling scope. In [HV06], Hendriks and Verhoef present a comparison between UPPAAL, POOSL, SymTA/S and MPA on an in-car radio navigation system based on worst-case response time analysis. This study confirms that UPPAAL finds accurate results, and the compositional analyses give larger values [HV06]. However, both studies point out that the UPPAAL analysis is time consuming and may suffer from state-space explosion.

1.2 Research Questions

The work of this thesis has been carried out under the Octopus project. This project aimed at developing methods and tools for system-level design of electromechanical systems. Examples of such systems are printers that perform under uncertain environmental conditions (e.g. resource breakdown, arrival of urgent jobs, job uncertain arrival times). This project was a cooperation between Océ Technologies, the Embedded Systems Institute and several academic research groups in the Netherlands. One of the main directions of the project was to explore techniques that allowed modeling and analysis of the datapath module of Océ printers. A datapath encompasses the complete trajectory of the image data from source (e.g. scanner) to target (e.g. printer).

The aim of this thesis is to expand performance evaluation of embedded systems using model-checking. In particular, we focus on the application of UPPAAL in datapath design. Other formal methods tools used in the project are CPN Tools and SDF.

Although a lot of research ([AME07, HBL+03]) has improved the computational capacity of Uppaal and scaled the tool up for industrial case studies, the datapath design brings a new research challenge. The majority of resources in a datapath modify their execution time at certain events that occur in the system. More precisely, while resource computation time is negligible, the transfer between resources and memory via a memory bus is a time consuming part. The transfer time changes according to the number of other competing resources that transfer data on the bus. We call these resources *dynamic*. In addition to this aspect, the level of detail and the large number of input jobs that the UPPAAL models should include could easily lead to state space explosion, a common problem in model checking. Therefore, a clever way of modeling is the first aspect to take into consideration when using UPPAAL for this industrial problem.

	Thesis Chapters				
Research Questions	Ch 2	Ch 3	Ch 4	Ch 5	Ch 6
Expressivity	✓		✓		
Complexity	✓	✓	✓	✓	
Efficiency				✓	
Uncertainty		✓	✓		
Robustness					✓

Table 1.1: Research questions and thesis outline.

The main research questions investigated in this thesis are:

- Expressivity: Is the UPPAAL modeling language expressive enough to model all relevant characteristics of the Océ datapath?
- Complexity: Can the UPPAAL verification engine handle the complexity of realistic printer datapath designs?
- Efficiency: How much does the modeling style influence the size of the state space?
- Uncertainty: Can UPPAAL analyze printing systems exposed to uncertain environmental conditions?
- Robustness: How robust are the schedules obtained with UPPAAL?

1.3 Contents of this Thesis

In this subsection, we give a brief description of the chapters of this thesis. Table 1.1 shows the places where we address the research questions mentioned in the previous section.

Chapters 2–5 are based on peer-reviewed published articles. Slight modifications were made to them to avoid redundancy. Furthermore, the layout and visual style of the papers was changed to improve consistency.

Throughout the thesis, we assume basic knowledge of timed automata theory and UPPAAL. In addition, in Chapter 6 we assume basic knowledge of the SMT-LIB language [RT06] that helps to understand the figures and script fragments.

Chapter 2

Chapter 2 presents an abstract description of an Océ datapath design problem. The printing system that we consider contains several resources, among others a USB, which has a limited bandwidth. Based on the features the system offers, several use cases can be defined. Scheduling of multiple concurrent jobs is a common problem in the printer design, each job having assigned a use case. In this sense,

we have chosen a challenging benchmark which we have represented with three modeling approaches: timed automata, CPN and SDF graphs. These models have been used to derive schedules on the benchmark, which we compare at the end of the chapter.

This chapter is based on the following paper:
[IKY+08] *G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. W. Vaandrager, M. Voorhoeve, S. de Smet, and L. J. Somers: Formal modeling and scheduling of datapaths of digital document printers. FORMATS 2008, LICS 5215, pages 170-187. Springer, 2008.*

Personal contributions: I have contributed to the scientific content and ideas, writing and reviewing process of this paper. In particular, I was the author of Sections 2.3.3 (except the description of the USB automaton) and 2.4 (the description of the UPPAAL results) and performed the experiments with UPPAAL.

Chapter 3

In the benchmark presented in Chapter 2, we assume that job arrival times are known. However, this is not the case in a realistic usage of an Océ datapath where jobs can arrive at any time. In addition, the benchmark contains many combinations of concurrent jobs which are unfeasible in reality, for example multiple scans at the same time. Therefore, in Chapter 3, we analyze a scenario that consists of two jobs: one that is continuously occupying the machine and another that has uncertain arrival times. The goal is to find solutions that provide an acceptable trade-off between the throughput of the continuous job and the latency of the other job. For this, we apply UPPAAL Tiga [Uppb, CDF+05] and describe the scenario as a two-player timed game.

Chapter 3 is based on the following paper:
[AHI+09] *I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F. W. Vaandrager: Adaptive scheduling of data paths using UPPAAL Tiga. QFM 2009, EPTCS 13, pages 1-11, 2009.*

Personal contributions: I contributed to the scientific content, text and reviewing process of this paper.

Chapter 4

In Chapter 4, we refine the case study analyzed in previous chapters by including more details regarding the Océ architecture such as a memory bus, separate memories and controller scheduling rules. The memory bus induces dynamic behavior in all image processing resources of the architecture. Moreover, we have implemented scheduling rules specific to Océ controllers. With these changes in the models, we analyze with UPPAAL the worst case latency of a job that has uncertain arrival time in a setting where the printer is concurrently processing an infinite stream of competing jobs.

This chapter is based on the following paper:
[IV10] *G. Igna and F. W. Vaandrager: Verification of printer datapaths using timed automata. ISoLA (2) 2010, LNCS 6416, pages 412-423. Springer, 2010.*

I am the main author of this paper: I contributed to the scientific content and ideas, to the construction of the models and I carried out all the experiments.

Chapter 5

Advances in the Octopus project have shown the need for an interface between the three modeling methods (i.e. CPN, timed automata and SDF) and a formal basis for the communication with the Océ engineers. As a result, *Parameterized Partial Order* (PPO) has been introduced to compactly represent large task graphs with repetitive behavior. In Chapter 5, we present a subclass of PPOs that can be efficiently translated into timed automata. We also prove that the transition system induced by the Uppaal models is isomorphic to the configuration structure of the original PPO. Moreover, we devise the notion of real-time task systems with tasks represented as PPOs. Lastly, we report on a series of experiments which demonstrates that the resulting Uppaal models are more tractable than the models of Chapter 2.

This chapter is based on the following papers:
[HIV12] *F. Houben, G. Igna, F. W. Vaandrager: Modeling task systems using parameterized partial orders. RTAS 2012: pages 317-327. IEEE, 2012*, and

F. Houben, G. Igna, F. W. Vaandrager: Modeling task systems using parameterized partial orders. International Journal on Software Tools for Technology Transfer, Springer-Verlag, pages 1-18, 2012.

Personal contributions: I contributed to the scientific content and ideas, writing and reviewing process of these papers. In addition, I improved the performance and the structure of the models generated for this chapter.

Chapter 6

Changes in the execution time of a resource are encoded in our Uppaal models as changes in the speed of the resource proportional with the number of concurrent resources that share the memory bus. However, we had to over-approximate the execution time of dynamic resources in the Uppaal models. In addition, Océ systems use a non-lazy scheduling policy i.e. if a resource is available when it is claimed to process a job, it immediately starts to process it. However, this might make some schedules invalid when used in a real machine. This happens because some job ordering from the Uppaal schedules cannot be anymore followed. Therefore, in Chapter 6 we investigate the validity of schedules obtained from the real-time task system representation of Chapter 5 when computing exact resource execution times. For this analysis we have employed Z3 [Z3 , dMB08], a state-of-the-art SMT (Satisfiability Modulo Theories) solver. Furthermore, we measure the robustness of the Uppaal schedules slightly varying the execution

times of some key dynamic resources. The robustness analysis is represented as a partial MaxSMT problem also solved with Z3.

Personal contributions: I was responsible for the main scientific ideas, tool building and experiments.

Chapter 7

In Chapter 7, we present a summary of the main contributions of this thesis and discuss some possible further directions.

Chapter 2

Case Study Description and Tool Comparison

We apply three different modeling frameworks — timed automata (UPPAAL), colored Petri nets and synchronous dataflow — to model a challenging industrial case study that involves an existing state-of-the-art image processing pipeline. Each of the resulting models is used to derive schedules for multiple concurrent jobs in the presence of limited resources (processing units, memory, USB bandwidth, ...). The three models and corresponding analysis results are compared.

2.1 Introduction

Industrial case studies are always difficult to model and analyze. On the one hand, too many details are costly to evaluate, but on the other hand, abstractions may leave out details that prove important for a meaningful analysis. Accordingly, some analysis tools are highly expressive, which hinders an exhaustive analysis, whereas others, which guarantee an exhaustive analysis, suffer from large amounts of memory consumed during analysis.

In this chapter, we introduce a case study on the design of Océ datapaths, that refers to scheduling of multiple concurrent jobs. We apply and compare three different modeling methods: Colored Petri Nets (CPN), Synchronous Dataflow (SDF) and Timed Automata (TA). These methods are known to serve the purpose of investigating throughput and schedulability issues. The objective of this chapter is to see whether the three methods can handle the particularities of this industrial case study, and to compare the corresponding models on criteria as quality, ease of construction, analysis efficiency, and predictability.

Petri Nets are used for modeling concurrent systems. They allow to both explore the state space and simulate the behavior of Petri nets models. *Petri Nets* are graphs with two types of nodes: *places* that are graphically represented by

circles, and *transitions* that are graphically represented by rectangles. Directed arcs are used to connect places to transitions and vice versa. Objects or resources are modelled by *tokens*, which are distributed across the places representing a state of the system. The occurrence of events corresponds to firing a transition, consuming tokens from its *input* places and producing tokens at its *output* places. CPNs (Colored Petri nets) are an extension where tokens have a value (color) and a time stamp. A third extension is hierarchy, with subnets depicted as transitions in nets higher in the hierarchy. We have used CPN Tools [JKW07, Jen92] as the simulation tool for the present case study.

Synchronous Dataflow (SDF) is widely used to model concurrent streaming applications on parallel hardware. An *SDF graph* (SDFG) is a directed graph in which nodes are referred to as *actors* and edges are referred to as *channels*. Actors model individual tasks in an application and channels model communicated data or other dependencies between actors. When an actor fires, it consumes a fixed number of tokens (data samples) from all of its input channels (the consumption rates) and produces a fixed number of tokens on all of its output channels (the production rates). For the purpose of timing analysis, each actor in an SDFG is also annotated with a fixed (worst-case) execution time. A timed SDF specification of an application can be analyzed efficiently for many performance metrics, such as minimum throughput guaranteed [GGS$^+$06], latency or minimum buffer sizes. Analysis tools, like the freely available SDF3 [SGB06], allow users to formally analyze the performance of those applications.

A number of mature model checking tools, in particular UPPAAL [BDL04], have been applied to the quantitative analysis of numerous industrial design problems. In particular, timed automata technology has been applied successfully to optimal planning and scheduling problems [HvdNV06, AAM06], and performance analysis of distributed real-time systems [HV06, PWT$^+$07].

Because timed automata are extensively used in this thesis, we present here the formal definitions on their syntax and semantics taken from [BY03]. For this, we assume a finite set of real-valued variables C for clocks and a finite alphabet Σ standing for actions, ranged over by $a, b, etc.$ Moreover, we define a clock constraint as a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$, for $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$ and $n \in \mathbb{N}$. Let $\mathcal{B}(C)$ denote the set of clock constraints. A *timed automaton* is tuple (N, l_0, E, I) where N is a finite set of locations, l_0 is the initial location, $E \in N \times \mathcal{B}(C) \times \Sigma \times 2^C \times N$ is the set of edges and $I : N \to \mathcal{B}(C)$ represents a set of location invariants. We write $l \xrightarrow{g,a,r} l'$ when $(l, g, a, r, l') \in E$. For defining the semantics of a timed automaton, we use functions called *clock assignments* that map C to the non-negative reals $\mathbb{R}_{\geq 0}$. Let u, v denote such functions. The notation $u \in g$ denotes that the clock values u satisfy the guard g. Also, for $d \in \mathbb{R}_{\geq 0}$, let $u + d$ denote the clock assignment that maps all $x \in C$ to $u(x) + d$. For $r \subseteq C$, let $[r \mapsto 0]u$ denote the clock assignment that maps all clocks in r to 0 and the value of the other clocks in $C \setminus r$ remains unchanged. The semantics of a timed automaton is a transition system where states

are pairs (l, u) and transitions are defined as follows:
– (discrete transitions) $(l, u) \xrightarrow{a} (l', u')$ if $l \xrightarrow{g,a,r} l'$, $u \in g$, $u' = [r \mapsto 0]u$ and $u' \in I(l')$,
– (delay transitions) $(l, u) \xrightarrow{d} (l, u + d)$ if $u \in I(l)$ and $(u + d) \in I(l)$ for a non-negative real $d \in \mathbb{R}_{\geq 0}$.

Outline of the chapter First we describe the case study. Second, we explain the timed automata, CPNs and SDF models. Then, we present a benchmark that we use as basis for the comparison of the three modeling approaches. At the end of the chapter we give conclusions and outline directions for future research.

2.2 Case Study Description

In addition to scanning, copying and printing, Océ systems perform a variety of image processing functions on digital documents, such as zooming or rotation. A *datapath* encompasses the complete trajectory of the image data from source (e.g. Scanner) to target (e.g. Printer). The architecture of the datapath studied in this chapter is shown in Figure 2.1. The system has two input ports: Scanner for local users and Data Store for remote users. The architecture contains image processing (IP) resources (i.e. ScanIP, IP1, IP2, PrintIP) and system resources such as memory or USB. Finally, there are two places where jobs can leave the system: Printer and Data Store.

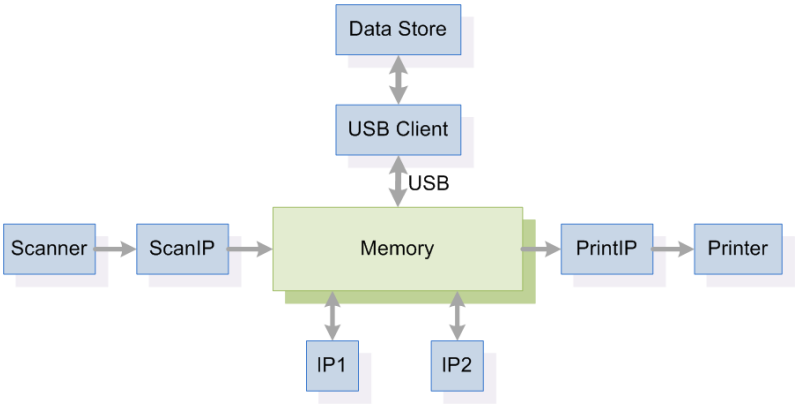


Figure 2.1: Datapath architecture of an Océ system.

The IP blocks can be used in different combinations depending on user preferences. This gives rise to different *use cases* (i.e. applications) of the system. A user sends her request as a *job* that contains a use case, a set of files, and image processing settings. Some examples of use cases are:

- **Direct Copy**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient, PrintIP[1]
- **Scan to Store**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ USBClient
- **Scan to Email**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
- **Process from Store**: USBClient $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
- **Simple Print**: USBClient $\rightsquigarrow$ PrintIP
- **Print with Processing**: USBClient $\rightsquigarrow$ IP2 $\rightsquigarrow$ PrintIP

In use case *Direct Copy*, a job is processed in order by the resources Scanner, ScanIP, IP1, and IP2, and then it is sent to the Data Store (via the USBClient) and printer (via PrintIP). In case of use case *Process from Store*, a remote job enters the system through the Data Store, it is further processed by IP1 and IP2, after which the result is sent back to the remote user via the USBClient and Data Store. The interpretation of the remaining use cases is similar.

The physical properties of the architecture allow a certain degree of parallelism. Scanner and ScanIP, for instance, may process a job in parallel. This is because ScanIP works fully streaming and has the same throughput as the Scanner. However, due to the characteristics of different resources, some additional constraints are imposed. For example, the image processing function that IP2 performs, forces IP2 to start processing a job only after IP1 has completed it. The dependency between ScanIP and IP1 is different. IP1 works streaming and has a higher throughput than ScanIP. Hence IP1 may start processing a job while ScanIP is also processing it, but situations where IP1 finishes earlier than ScanIP must be avoided.

In addition to the image processing resources, two other system resources that may be scarce are memory and USB bandwidth. Execution of a job commences only after allocation of the memory required for the completion of the entire job. Each resource requires a certain amount of memory to process a job. This memory can be released once the computation has finished and no other resource need it. Memory management is one of the decisive factors in determining the throughput and efficiency of the datapath. Another critical resource is the USB because is has limited bandwidth. USB serves as a bridge between the USBClient and memory. Moreover, it may be used for both uploading and downloading data to/from the Data Store. At any instance of time, at most one job may upload data, and similarly at most one job may download data. Uploading and downloading may take place concurrently. If the transfer is unidirectional, then it takes place at a rate of *high* MByte/s. Otherwise, it takes place at a slightly lower rate

[1] If A $\rightsquigarrow$ B occurs in an use case, then the start of the processing by A should precede the start of the processing by B. In addition, if A, B occurs in an use case, then A and B may run in parallel.

of *low* MByte/s.[2] This is referred to as the *dynamic* USB behavior. The *static* USB behaviour is the one in which the transmission rate is always *high* MByte/s.

The main challenge addressed in this case study is to compute efficient schedules that minimize the execution time for jobs and realize a good throughput. A related problem is to determine the amount of memory and USB bandwidth required, so that these resources would not become bottlenecks in the system performance.

2.3 Modelling Approaches

2.3.1 CPN

In the Octopus project, the Petri Net approach takes an architecture-oriented perspective to model the Océ case study. In addition to the system characteristics, the model includes scheduling rules (e.g. First Come First Served is used when jobs enter the system) and is used to study the performance of the system through simulation. Each resource is modeled as a subnet. Since the processing time for all resources, except the USB, can be calculated before they start processing a job, the subnet for these resources looks like the one shown in Figure 2.2. The

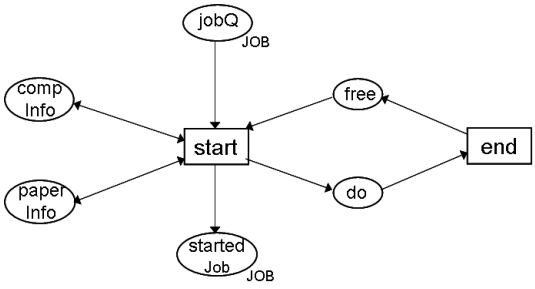


Figure 2.2: Hierarchical subnet for the resources Scanner, ScanIP, IP1, IP2 and PrintIP.

transitions *start* and *end* model the beginning and completion of a job, while the places *free* and *do* reflect resource state. In addition, there are two places that characterize the subnet of each resource: *compInfo* and *paperInfo*. The place *compInfo* contains a token with information required by a resource before starting a new job, namely the resource ID, processing speed and the recovery time. The place *paperInfo* contains information on the number of bytes a particular resource processes for a specific paper size. The values of the tokens at places *compInfo*

[2] Approximately, *low* is 75% of *high*. The reason why it is not 50% is that the USB protocol also sends acknowledgment messages, and the acknowledgment for upward data can be combined with downward data, and vice versa.

and *paperInfo* remain constant after initialization and govern the behavior of a resource.

Place *jobQ* in Figure 2.2 contains tokens for the jobs that are waiting for resources. The color of a token of type *Job* contains information about the job ID, the use case and paper size of the job. Hence, a resource can calculate the time required to process a job from the information available in the *Job* token, and the tokens at the places *compInfo* and *paperInfo*. Once the processing is completed, the transition *end* places a token at the place *free* after a certain delay, governed by the recovery time specific to each resource, thus determining when a resource can begin to process the next job available.

Figure 2.3 shows an abstract view of the entire model constructed for the Océ case study. New jobs can be created using the *Job Generator* subnet, which are placed as input to the *Scheduler* subnet at the place *newJob*. The *Scheduler* subnet models the scheduling rules, memory management rules and routes each job from one resource to the next based on job use case. In this model, the scheduling rules are viewed as being global to the system and not local to any of the resources.

The *Scheduler* picks a new job that enters the system from the place *newJob* and estimates the amount of total memory required for executing the job. If enough memory is available, the memory is allocated (the memory resource is modeled as an integer token in the place *memory*) and the job is scheduled for the first resource in the use case by placing a token of type *Job* in place *jobQ*, which will be consumed by this resource. When a resource starts processing a job, it immediately places a token in the *startedJob* place indicating this event. The *Scheduler* consumes this token to schedule the job to the next resource in its use case, adding a delay that depends on the resource that has just started, the next resource in the use case and the dependency between resources explained in Section 2.2. Thus the logic in the *Scheduler* includes scheduling new jobs entering the system (from place *newJob*) and routing the existing jobs through the resources according to the corresponding use cases. As mentioned above, the *Scheduler* subnet also handles the memory management. This includes memory allocation and release for jobs that are executed.

USB The USB model is different from that of the other resources since the time required to transmit a job (upstream or downstream) is not constant and is influenced by other jobs that may be transmitted at the same time.

The CPN model of the USB works by monitoring two events: (1) a new job joining the transmission, and (2) completion of a job transmission. Both events influence the transmission rates for any other jobs on the USB, and hence determine the transmission times for the jobs. In the model shown in Figure 2.4, there are two transitions *join* and *update*, and two places *trigger* and *USBjobList*. The place *USBjobList* contains the list of jobs that are currently being transmitted over the USB. We require that at any point in time, there are at most two jobs active in this list: one uploaded and one downloaded. Apart from information about

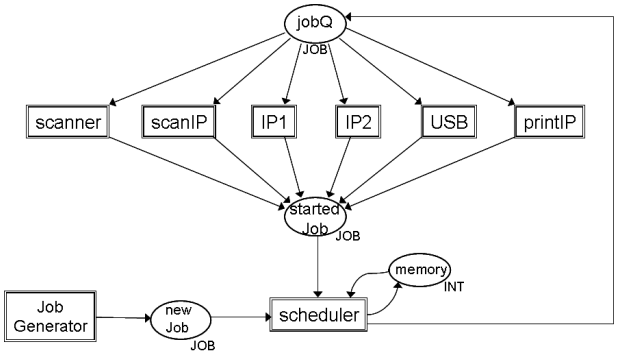


Figure 2.3: Architectural view of the CPN model.

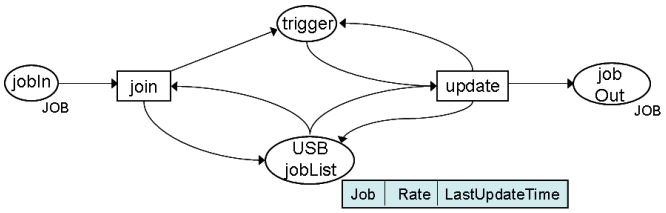


Figure 2.4: CPN model for the USB.

each job, it also contains the transmission rate currently assigned, the number of bytes remaining to be transmitted and the last time of update for each job. The transition *join* adds a new job at place *jobIn* that requests the USB (if it can be accommodated) to *USBjobList*, and places a token at place *trigger*. This enables the transition *update* that checks the list of jobs at place *USBjobList* and reassigns the transmission rates for all the jobs according to the number of jobs transmitted over the USB. The *update* transition also recalculates the number of bytes left for transmission for each job since the last update, estimates the job that will finish next and places a timed token at *trigger*, so that the transition *update* can remove the jobs whose transmissions are completed. The jobs whose transmission are complete are placed in the place *jobOut*. Thus the transition *join* catches the event of new jobs joining the USB and the transition *update* catches the event of jobs leaving the USB, which are critical in determining the transmission time for each job.

2.3.2 Synchronous Dataflow

In the Synchronous Dataflow (SDF) approach, we choose to model the Océ system from an application-oriented perspective. In contrast to the two earlier approaches, we take a compositional approach that targets analysis efficiency for applications.

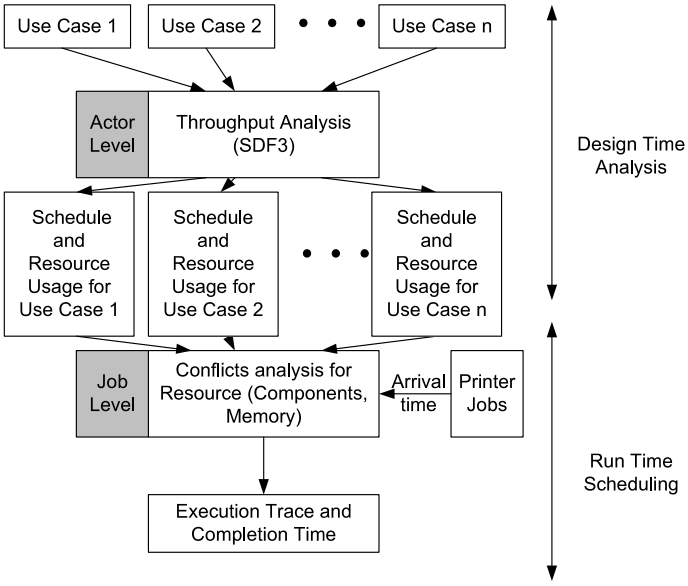


Figure 2.5: Job scheduling using SDF models.

Since SDF is particularly well suited to optimize throughput for streaming applications, we focus on the scheduling problem for job sequences consisting of jobs with many iterations per use case (e.g. a Direct Copy job having a 500-page file). Each use case of the case study described in Section 2.2 is modeled as an SDF graph. Architecture information is included by annotating graph actors with resource usage information. The scheduling problem is tackled via a 2-phase methodology (see Figure 2.5):

1. the design time analysis, where we apply SDF3 [SGB06] to generate a throughput-optimal schedule per use case,

2. the runtime scheduling, in which the schedule is adjusted based on arrival times of jobs. In this phase, the schedule also takes the system constraints (e.g. number of resources available, memory amount) into account.

This two-phase scheduling approach guarantees job completion times for arbitrary job sequences. It avoids the complexity of analyzing all the details of a job sequence at runtime, which is infeasible in general, sacrificing some performance that might be obtainable via global optimization. The approach can be seen as an instance of the Task Concurrency Management method of [WMY01], providing predictability by the use of SDF as a modeling formalism.

Use Case Modeling In this approach, computations performed by resources are modeled as actors. Actors are annotated with execution times and resource usage information. The delays imposed by resource dependencies given in Section

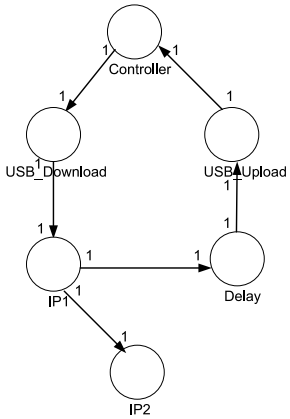


Figure 2.6: SDF model of the ProcessFromStore use case

2.2 are also explicitly modeled by means of actors. The USB communication is split into two actors: USB_Download and USB_Upload. The execution times of the USB actors are approximated by always assuming high bandwidth availability. Thus, use case analysis can be decoupled from job scheduling, sacrificing some accuracy in exchange for analysis efficiency. Figure 2.6 shows the SDFG of the ProcessFromStore use case, actor production and consumption rates are equal to one.

Job Scheduling and Completion Time Analysis The analysis of an SDF model of a single job is straightforward. By ensuring composability through virtualization of resources (every job gets its own, private share of the resources claimed), multiple jobs can also be analyzed efficiently. However, as the resources can be shared between jobs, the existing techniques to analyze multiple jobs cannot be applied directly to the analysis of the Océ case study. The challenges faced with this modeling approach are how to model the behavior of concurrent jobs in a non-virtualized way and how to calculate the completion time of these jobs.

In order to analyze the case study without virtualization, we make some assumptions. We conservatively assume that the resources needed by actors are claimed at the start of a firing and released at the end of firing. In addition, the claim and release of a resource like the memory may happen in different actors. A waiting job can start if all resources required can be reserved. Resource reservation ensures that the execution time of all job tasks are fixed. As already mentioned, USB bandwidth is always assumed to be high. These assumptions make the system efficiently analyzable by limiting its dynamic behavior, and allow the two-phase scheduling approach explained above. The first phase concerns actor level and uses throughput-optimal self-timed execution (data-driven, every actor fires as soon as it is enabled) as scheduling strategy for a single use case. Resource usage of each use case is calculated using this self-timed schedule. The second phase concerns job scheduling. Jobs are served in a First Come First Served (FCFS) way. If the

resources required by a new job cannot be ensured at arrival time, the new job has to be postponed until the resources are available. Jobs can still be pipelined, overlapping in time, as illustrated below. Two types of resources, i.e. disjunctive and cumulative, are considered in detail. IP resources are an example of disjunctive resources that can only be used by one job at a time, while shared memory is an example of a cumulative resource that can be used by many jobs as long as the total usage does not exceed the maximum amount available.

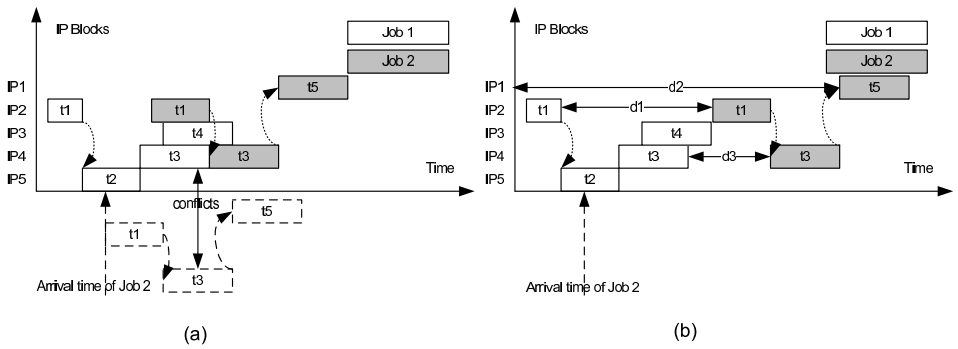


Figure 2.7: Scheduling concurrent jobs.

Figure 2.7 shows the scheduling of two jobs (Job 1 and Job 2) of different use cases with a resource conflict (ignoring the memory usage). The self-timed schedule and its resource usage are computed in phase one using the throughput analysis algorithm in SDF3. The work is done off-line, avoiding computation work at runtime. In phase two, the scheduler computes the earliest start time of the second job based on the results of phase one and runtime information. According to Figure 2.7(a), Job 2 cannot start when it arrives due to a resource conflict on IP4. In order to maximize the resource usage, Job 2 has to start at a point that ensures that IP4 can be used by Job 2 as soon as it is released by Job 1. Figure 2.7(b) illustrates how to compute this specific point. When Job 1 starts its execution, all resources and memory it needs are reserved. A system resource usage table is kept to store the release time of resources. When Job 2 arrives, we initially assume that it starts at the end of the last actor of Job 1 (t4 of Job 1 in Figure 2.7(b)). Then, we calculate the time distance between the current release time of resources and the reservation of those resources for Job 2. The start point then equals the end time of Job 1 minus the minimum of the computed distances (d3 in Figure 2.7(b)). In order to analyze memory conflicts, we store the memory usage of each use case as a list of pairs: (memory usage, time interval). Figure 2.8 shows how to update the memory when a new job starts (q_i represents the amount of memory needed at time t_i). As a single job always fits in the memory, we only need to consider the overlap between a new job and any running jobs that occupy memory. We define a memory usage interval $MI_i = (q, [t_i, t_{i+1}))$ to represent that in the time interval from t_i and t_{i+1} the memory usage equals q. Figure 2.8 illustrates that

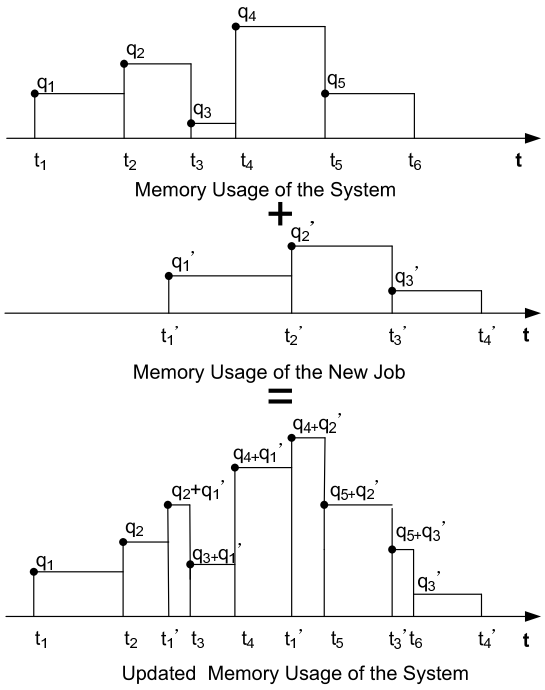


Figure 2.8: Update memory usage.

the updated memory usage is determined by the memory usage intervals of the system and those of a new job. We verify memory by intersecting intervals and calculate the time a job may need to be postponed. Assume that the maximum amount of memory available is q_c, and $MI_2 = (q_2, [t_2, t_3))$ is already occupied and $MI_1' = (q_1', [t_1', t_2'))$ is claimed by a new job with $t_1' < t_3$. If $q_2 + q_1' < q_c$, the new job can be accommodated. Otherwise, the new job is delayed for $t_3 - t_1'$ time units. We repeat this check until there is no interval left where the memory occupied exceeds the maximum memory available. Other resources, like the USB, can be treated in the same way.

2.3.3 Timed Automata

In the timed automata approach, we have created six job templates, a template for each use case a job may use. In order to create a job, we instantiate one of these templates based on the use case required. In addition, each resource is modeled as a separate timed automaton, except for the memory, which is defined as a shared variable.

All image processing resources follow the same behavioral pattern: a job claims a resource, when the resource becomes available it processes the job for a period of time depending on job size, after which a recovery phase occurs. In the recovery phase, resources perform some initialization steps e.g. when the scan has finished,

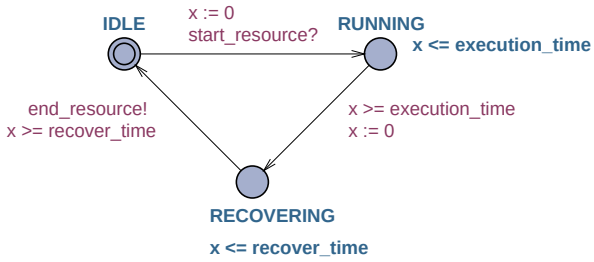


Figure 2.9: Resource template.

the scanner arm is moved back to the start position.

Figure 2.9 shows the template each resource automaton instantiates. Initially a resource is in location IDLE. When a resource is claimed (i.e. a job automaton synchronizes with the resource automaton via channel start_resource), the edge to location RUNNING is taken. On that edge, we can also see a reset of clock x. Clock x records in the first phase the time spent in the execution phase, and then the time spent in the recovery phase. Variable execution_time specifies the total time a resource needs for the execution of a job. Location RUNNING is labeled with the invariant x <= execution_time, which specifies that the value of clock x must be smaller or equal to execution_time whenever location RUNNING is active. After execution_time time units elapse, the edge to location RECOVERING is taken. This edge is labeled with the clock guard x >= execution_time, which is the enabling condition for the edge. The invariant of location RUNNING and this guard ensures that the transition is taken exactly after execution_time time units. Location RECOVERING is active for exactly recover_time time units. The edge to location IDLE may be taken afterwards. Both execution_time and recover_time are parameters that are set by a job automaton before the job claims a resource.

Figure 2.10 gives the automaton corresponding to a job that uses the *Direct Copy* use case. We can observe in the figure that the claim of the first resource occurs only after memory is reserved for all resources required in the use case. In addition, memory release happens at the end of IP2 and at the end of a job. On each edge that claims a resource, the resource execution time is set. The figure also illustrates the way we model the parallel activities between the resources IP2, USBClient and PrintIP. Due to the sequential way of modeling in Uppaal, we have added a third party automaton between a job automaton and a resource automaton that run in parallel (see Figure 2.11). Each extra automaton registers the claim (channel resource_claimed) to the resource automaton, then it waits for the resource automaton to become available. When it is available, it sends the request to the resource automaton through channel start_resource). On the completion of the processing (channel end_resource fired), it sends an end event to the job automaton (via channel resource_released). Channel resource_claimed in Figure 2.11 is a parameter for the arrays of channels usb_transfer_claimed and print-

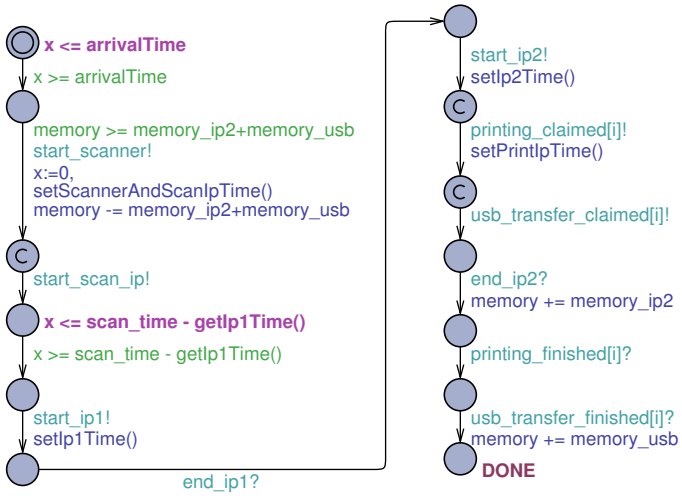


Figure 2.10: *Direct Copy* template.

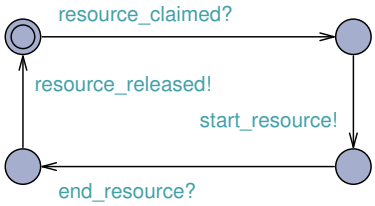


Figure 2.11: Interface automaton for modeling parallel activities.

ing_claimed that ensure the synchronization of *Direct Copy* job automata (see Figure 2.10) with the third party automata that register a claim request for the USB or PrintIP, respectively. Similarly, the arrays of channels usb_transfer_finished and printing_finished ensure in reality the synchronization of job automata with third party automata when the latter are announced that the USB or PrintIP have finished a job.

USB A challenging aspect in modeling the architecture was the USB because of its dynamic behavior. Firstly we modeled it as a linear hybrid automaton as it can be seen in Figure 2.12. Linear hybrid automata [ACH+95] are a slight extension of timed automata in which besides clocks, they also allow other continuous variables with constant rates that may depend on the location. In the automaton of Figure 2.12, there are two continuous variables: up and down, modeling the amount of data that needs to be uploaded and downloaded, respectively. In the initial state the bus is idle (derivatives $\dot{up}$ and $\dot{down}$ are equal to 0) and there are no data to be transmitted (up = down = 0). When upload starts (event start_up?), variable up is set to U, the number of MBytes to be transmitted, and the derivative

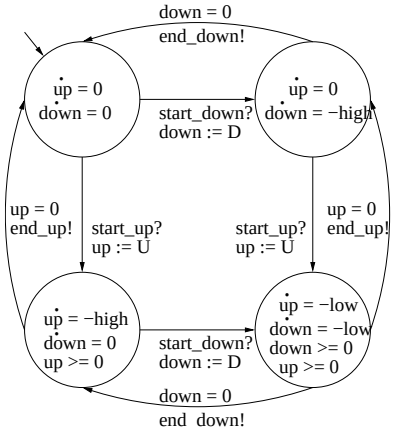


Figure 2.12: Linear hybrid automaton model of the USB bus.

$\dot{\mathsf{up}}$ is set to $-\mathsf{high}$. Upload ends ($\mathsf{end\_up!}$) when there are no more data to be transmitted, i.e., location in which down is set to D and both $\dot{\mathsf{up}}$ and $\dot{\mathsf{down}}$ are set to $-\mathsf{low}$. There are dedicated model checkers for linear hybrid automata, such as HyTech [HHWT97], but the modeling languages supported by these tools are rather basic and the verification engine is not sufficiently powerful to synthesize schedules for our case study. The problem faced here is that this type of hybrid behavior cannot be easily modeled in UPPAAL. We could have used stopwatch automata (i.e. timed automata extended with stopwatches) which are as expressive as linear hybrid automata [CL00], but they are not very practical to encode the USB behavior[3].

We experimented with several timed automaton models that approximated the hybrid model. In the simplest approximation, we postulate that the data rate is high, independently of the number of users. This behavior can simply be modeled using two instances of the resource template of Figure 2.9. Our second "dynamic" model, shown in Figure 2.13, overapproximates the computation times of the hybrid automaton. Clock x records the time since the start of the latest change in the transmission rate. Integer variables up and down give the number of MBytes left for transmission. If an upward transmission starts in the initial state, up is set to U and x to 0. Without concurrent downward traffic, transmission will end at time $\mathsf{divide}(\mathsf{up}, \mathsf{high})$[4]. Suppose that downward transmission starts somewhere in the middle of upward transmission, when clock x has value t. At this point still $\mathsf{up} - \mathsf{high} \cdot \mathsf{t}$ MByte needs to be transmitted. The problem we confront

[3]Note 2013: In 2011, priced timed automata have been extended with stochastic semantics which offers the possibility to define clocks that may evolve with different rates [DLL+11, DDL+12]. This type of automata represents a promising alternative to explore for this case study.

[4]Since in timed automata we may only impose integer bounds on clock variables, we use a function $\mathsf{divide}(a, b)$, which gives the smallest integer greater or equal to $\frac{a}{b}$.

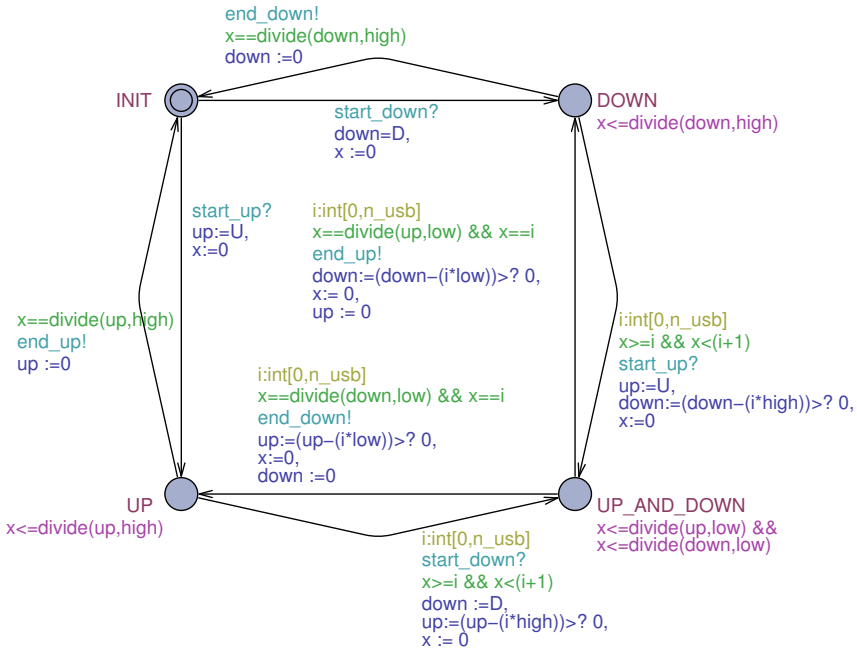


Figure 2.13: Second timed automaton model of the USB bus.

here is that in UPPAAL we cannot refer to the value of clocks in assignments to integer variables. However, and this is an interesting trick, using the select statement[5] we may infer the largest integer i satisfying $i \leq t$. We update up to the maximum of $up - high \cdot i$ and 0, which is just a small over-approximation of the amount of data remaining to be transmitted, and reset x. The other transitions are specified in a similar style.

The UPPAAL verification engine is able to compute the fastest schedule for completing all jobs (without any a priori assumption about the scheduler such as first come first served). However, for more than 6 jobs, the computation times increase sharply due to state space explosion. The state space explosion problem can be alleviated by declaring urgent (some of) the start_resource channels. In this way we impose a *non-lazy* scheduling strategy in which a resource is claimed as soon as it has become available and some job needs it. This strategy reduces UPPAAL computation times from hours to minutes. However using this strategy we have lost the guarantee of obtaining optimal schedules.

[5] Adding a select statement $i : int[0, n\_usb]$ to a transition effectively amounts to having a different instance of the transition for each integer i in the interval $[0, n\_usb]$.

2.4 Result Comparison

This section presents a comparison of the optimal schedules obtained with the three approaches for the Océ datapath presented in Section 2.2. We have chosen a common arrival sequence of seven single-page jobs shown below:

JobID	Use case	Arrival time(s)	Memory required(MB)
A6	Print with Processing	0	12
A7	Process from Store	0	24
A2	Scan to Email	1	48
A3	Scan to Store	1	36
A5	Print with Processing	1	12
A1	Process from Store	2	24
A4	Process from Store	3	24

The first part of this section reports on the results obtained via the three approaches where the USB has a static behavior and the last part reports on the case where the USB is dynamic.

Static USB behavior

CPN approach Figure 2.14 gives the best schedule obtained with CPN Tools which has a total completion time of 24 seconds. Even though jobs A6 and A7 arrive at the same time and request *USBdown* simultaneously, job A6 is chosen non-deterministically to use *USBdown* first at time 0. Such resource contentions can also be observed when job A5 waits for the *PrintIP* to become available as it is processing job A6, even though job A5 can be processed by *PrintIP* as soon as the *USBdown* is completed. Job sequence is also influenced by the memory available in the system. Even though job A2 arrives before jobs A3, A5, A1 and A4, its execution commences only at time 15, as shown in Figure 2.14, because until this time, there is less memory available than what is required for job A2.

SDF approach In the SDF approach, jobs are served in an FCFS fashion. The job order is determined by the arrival time of each job. If more jobs arrive at the same time, the order is determined nondeterministically. For the given benchmark, there are 12 possible orders for the 7 jobs ($2! \cdot 3! \cdot 1 \cdot 1 = 12$) and the best schedule has the shortest completion time of 27 seconds (see Figure 2.15). From the figure, we can see that job A2 is postponed until the claimed memory has become available.

TA approach For the analysis with UPPAAL of the two cases, we have asked UPPAAL for the fastest trace (option -t2) that satisfies the following query:

```
E<> A1.DONE && A2.DONE && A3.DONE &&
    A4.DONE && A5.DONE && A6.DONE && A7.DONE
```

UPPAAL computed an optimal schedule with a completion time of 22 seconds, displayed in Figure 2.16. Figure 2.17 illustrates the memory usage of this schedule. At time 10 memory is released as job A2 is completed, and immediately jobs A3

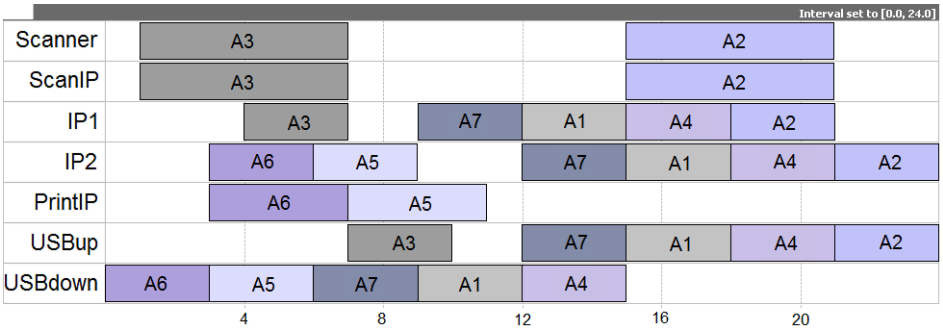


Figure 2.14: Execution chart for the CPN model with static USB behavior.

and A5 can start. Figure 2.16 shows that IP2 is the critical resource, and that it is optimally used.

The three schedules obtained are surprisingly different, UPPAAL being the only one that finds the optimal schedule. For the simulation-based approach with CPN Tools, the result depends on the simulation time, and longer simulations lead to better schedules. The SDF approach follows a strict FIFO scheduling and hence the total completion time of jobs is the largest. However, it is the only approach that is compositional, and hence it is expected to scale better to larger job sequences.

Dynamic USB behavior

CPN approach As shown in Figure 2.18, the total completion time is 25.5 seconds as against 24 seconds for the static USB behavior. Analyzing the simulation results of the static and dynamic USB behavior, the difference in completion time is caused by the change in transmission rates of the USB.

TA approach The result for the dynamic USB model is depicted in Figure 2.19. The total completion time is 25 seconds. The figure shows that the difference between this result and the one for the static model is caused solely by the USB dynamic behavior. As a result, some actions are right-shifted, but the order between jobs is the same as obtained for the static case. We can also see in the figure that the USB upload and download synchronize perfectly, meaning that the over-approximation introduced in the Uppaal model is not an issue here. As a result, we claim that this is the optimal schedule for the dynamic behavior.

2.5 Conclusions and Future Work

We have applied three prominent state-based modeling frameworks —UPPAAL, Colored Petri Nets, and Synchronous Dataflow— to an industrial case study, and managed to compute schedules for a representative benchmark. Our preliminary

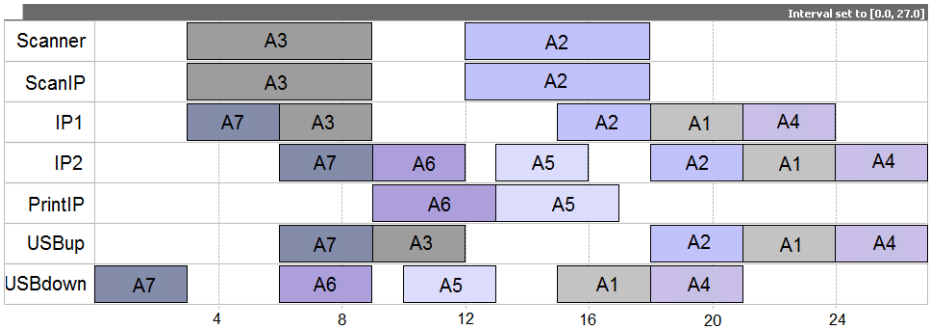


Figure 2.15: Execution chart for the SDF model with static USB behavior.

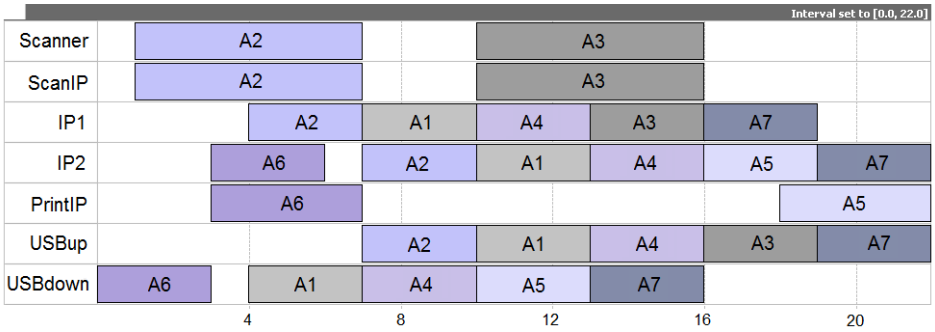


Figure 2.16: Execution chart for the TA model with static USB behavior.

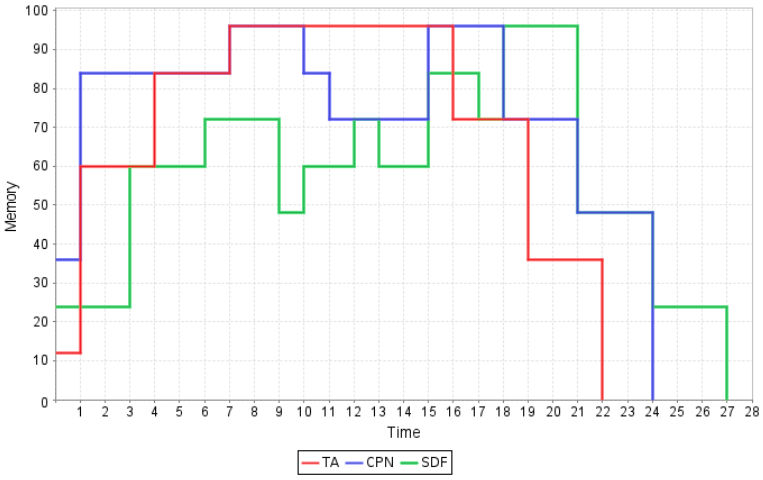


Figure 2.17: Memory usage for static USB model.

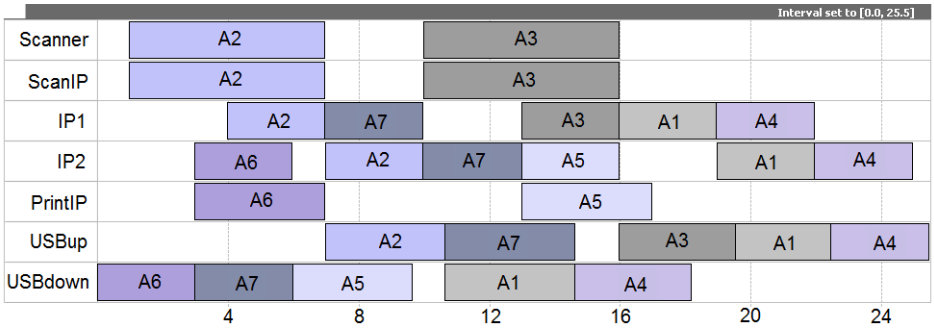


Figure 2.18: Execution chart for the CPN model with dynamic USB behavior.

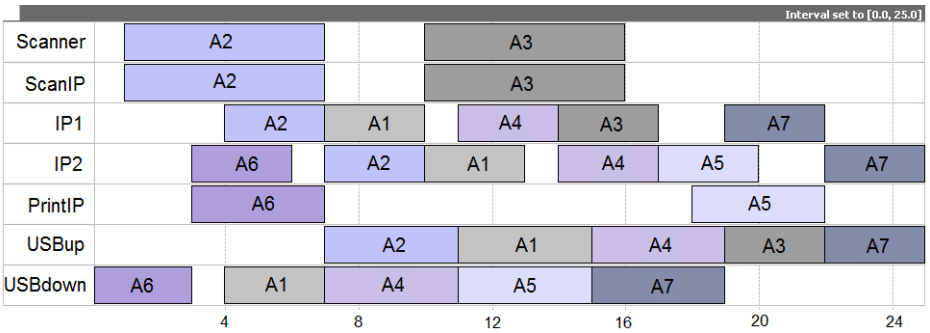


Figure 2.19: Execution chart for the TA model with dynamic USB behavior.

conclusion is that Colored Petri Nets provide the most expressive modeling framework, whereas UPPAAL currently appears to be the most powerful tool for finding (optimal) schedules. However, this case study pushes UPPAAL to its limits and since the SDF approach, which is the only compositional one, and therefore more scalable, it is certainly possible that it will outperform UPPAAL on larger benchmarks.

We have not embarked on the enterprise to formally relate the three different models. However, we can confirm the result of [HV06] that the construction of models of the same system using different tools helps to find bugs in the models, and thus contributes to improving the quality of the models.

From a modeling perspective, a very interesting feature in the Océ case study is definitely the USB bus. We consider surprising that timed automata are able to deal so well with what at first sight appears to be a hybrid phenomenon. The select statement from UPPAAL is crucial in defining this model.

The benchmark employed in this chapter is, however, larger than the normal usage of an Océ printing machine. A common scenario includes not more than two jobs coming from two different inputs. In the next chapter, we employ Uppaal Tiga on this case study to search for optimal schedules where two concurrent jobs

are modeled as a two-player timed game.

Chapter 3

Dynamic Scheduling with UPPAAL-TIGA

We apply UPPAAL-TIGA to automatically compute adaptive scheduling strategies for the case study presented in the previous chapter.

3.1 Introduction

Scheduling concerns the allocation of resources to activities over time in order to achieve some goals. Scheduling problems occur in many different domains and a vast amount of research has been carried out in this area. However, scheduling is usually seen in the research literature as a function of known, perfect inputs: the set of jobs, their arrival times, resource capacities, the duration of activities, and other characteristics of the problem are assumed to be known and static [DB00]. Nevertheless, in practice, scheduling processes are driven by uncertainty [MBS89, MW99]. This uncertainty may arise due to various sources (machine breakdown, unexpected arrival of new jobs, modification of existing jobs, uncertainty of task durations, etc.). McKay et al. [MSB98] claim that the dynamic characteristics of some real-world scheduling environments render the bulk of existing solution approaches for the job shop problem unusable when applied to practical problems. The problem of computing *optimal* scheduling strategies in practical settings with uncertainty is still open. The present chapter aims to address this problem using UPPAAL-TIGA [Uppb], an extension of UPPAAL [BDL04], the well-known timed automata model checker.

UPPAAL-TIGA implements the first efficient on-the-fly algorithm for solving games based on timed game automata with respect to reachability and safety properties [BCD+07, CDF+05]. In UPPAAL-TIGA, systems are specified through a network of timed game automata [MPS95]. These are timed automata in the sense of [AD94] where edges are marked as either controllable or uncontrollable.

This defines a two-player game with on the one side the *controller* (mastering the controllable edges) and on the other side the *environment* (mastering the uncontrollable edges). Winning conditions of the game are specified through TCTL formulas and state for instance that, irrespective of the strategy used by the environment player, the system player can always reach (or always avoid) certain states. In a scheduling context, uncertainty can be modeled using uncontrollable edges. UPPAAL-TIGA is then able to synthesize strategies for controlling the system such that certain scheduling objectives are met regardless of the timing of the uncontrollable edges.

In order to demonstrate the practical usefulness of UPPAAL-TIGA for solving scheduling problems with uncertainty, we have applied the tool to the Océ case study presented in the previous chapter and that concerns the scheduling of a state-of-the-art image processing pipeline of a printer. An initial version of this scheduling problem has also been described there and analyzed using three different modeling frameworks: timed automata (UPPAAL), colored Petri nets and synchronous dataflow graphs. However, none of these models incorporated uncertainty and in particular it was assumed that job arrival time is known in advance, which in reality does not hold. Therefore, in this chapter we employ UPPAAL-TIGA to search for optimal scheduling strategies for scenarios in which the arrival time of certain jobs is unknown. Other industrial applications of UPPAAL-TIGA deal with the synthesis of controllers [JRLD07, CJL+09, FLT09, OFCF11] and autonomous robot synthesis [AAG+07], which makes our analysis among the first applications to an industrial scheduling problem.

The rest of this chapter is organized as follows. First, we compare the way in which UPPAAL and UPPAAL-TIGA deal with uncertainty. Second, we present the modifications on the UPPAAL model of the previous chapter to represent uncertainty in job arrival times. Then, we present the results of the analysis with UPPAAL-TIGA and discuss how they can be used to improve real printers/copiers. Finally, we present some conclusions and directions for further work. The models described in this chapter are available on-line at the following address:

`http://www.cs.ru.nl/ita/publications/papers/fvaan/TigaOce`.

3.2 Comparison UPPAAL – UPPAAL-TIGA

Uncertainty of job arrival times can be modeled in UPPAAL using non-determinism. However, UPPAAL allows users to specify search options that dictate the strategy with which UPPAAL verification engine performs state space search. In this way, UPPAAL can be tuned such that when searching for an optimal schedule, it chooses job arrival times that allow for the optimal schedule. Moreover, since the UPPAAL tool is based on timed automata, rather than timed game automata, we cannot introduce a distinction between controllable and uncontrollable delays.

As mentioned earlier, UPPAAL-TIGA defines a two-player game: the controller,

mastering the controllable edges against the environment, mastering the uncontrollable edges. This type of game can be used to generate strategies that are independent on job arrival time. Intuitively, we can perceive UPPAAL as having one player, the controller, which has control over all edges, whereas UPPAAL-TIGA employs concepts from game theory to separate environment actions from system actions. The player representing the environment controls the arrival time of unpredictable jobs. The goal of this player is to prevent the controller from winning by activating unpredictable actions at the worst possible time.

3.3 Scenario

Although several use cases can be defined for the Océ datapath as presented in Chapter 2, a common scenario is the one in which a user makes a copy of a document and another user sends a job from a remote location. Hence, the scenario that we analyze in this chapter consists of a series of Direct Copy jobs that is interrupted from time to time by sporadic Print with Processing jobs. The Print with Processing jobs should be processed within a reasonable time, but the Direct Copy jobs should not be delayed for too long. Therefore, our goal is to find a strategy that can deal with the unpredictable nature of the Print with Processing job and guarantees an acceptable trade-off between the throughput of the Direct Copy jobs and the latency of the Print with Processing jobs.

3.4 Model Description

The case study presented in Chapter 2 is the starting point for the application of UPPAAL-TIGA. As in the UPPAAL model, each job and each resource is described in the corresponding UPPAAL-TIGA model as a separate automaton, except for memory which is simply modeled as a shared variable. Moreover, an edge is introduced in each job template from the last location to the initial location for modeling jobs with multi-page files or large batches of jobs that use the same use case. Furthermore, we have implemented the so-called *non-overtaking* scheduling rule to ensure that no two jobs of the same use case or two pages of the same job compete for resources. As described in [BBHM05], this rule reduces the state space without loss of the optimal solutions.

Figure 3.1 presents the model of the Print with Processing job. The first edge is uncontrollable and encodes the unpredictable arrival time of this job. The latency (i.e. the total time in which a job is completed) of this job is recorded by clock timeSinceArrival which is reset after each job completion.

Job throughput is measured with an Observer automaton (see Figure 3.2). Assume a job that contains a multi-page file or multiple single-page jobs of the same use case. Clock x in the figure measures the time elapsed between the completion of two consecutive pages. At the completion of a page, the corresponding job

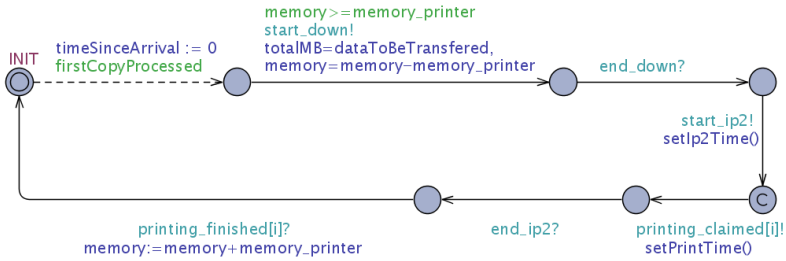


Figure 3.1: The automaton of Print with Processing job in UPPAAL-TIGA.

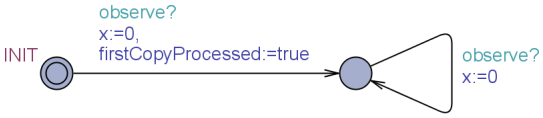


Figure 3.2: Observer template.

automaton synchronizes with an Observer automaton via the observe channel. The Observer automaton consists of two locations. The first monitors the completion of the first page and the second the time distance between the completion of two consecutive pages. Usually, the first page has a different throughput requirement than the subsequent pages. For our scenario, when the first page of the DirectCopy job is done, the flag firstCopyProcessed will be set to true. This flag ensures that the Print with Processing jobs arrive after the completion of the first Direct Copy page (see Figure 3.1). Usually, after a number of pages processed (e.g. one in our case), a recurrent part occurs, which is of interest in our analysis. For this reason, we have chosen to measure the Print with Processing job latency after the completion of the first page of the Direct Copy job.

Urgent Channels

We observed that even the verification of a relatively small number of jobs becomes infeasible. To reduce the complexity of this model, we made *urgent* the channels between job and resource automata. This implies that as soon as a job claims a resource which is in the idle mode, the resource processes it immediately (see the notion of non-lazy scheduling of [AAM06]). This reduces the number of possibilities during the verification process. However, the channels of resource automata whose resources are shared should not be made urgent. In our scenario, there are only two resources shared between the two use cases: the printer and IP2. As a

consequence, their corresponding channels are non-urgent. The reason why these channels should not be urgent is illustrated by the following example: assume that a Direct Copy job and a Print with Processing job start at the same time. According to the Print with Processing use case, this job is the first that claims the printer. If the printer channel is urgent, the Print with Processing job immediately occupies the printer. Accordingly, the Direct Copy job which claims the printer a short time after has to wait, which would restrict the controller choices.

Winning Condition

We assume two infinite streams of Direct Copy jobs (DC) and one infinite stream of Print with Processing jobs (PP). This means that each job contains a file with an infinite number of pages, but, at any point in time, at most two pages of the DC jobs and one of the PP jobs can be present in the system. We have observed that unless no PP job is allowed, throughput of the DC jobs does not increase when increasing the number of DC jobs. Our goal is to find a trade-off between high throughput of DC jobs and low latency of the PP jobs. Accordingly, we ask UPPAAL-TIGA the following query:

```
control:A[]
(DC_OBSERVER.INIT imply DC_OBSERVER.x <= FIRST_DC_TIME)  &&
(!DC_OBSERVER.INIT imply DC_OBSERVER.x <= DC_TIME)  &&
(!PP.INIT imply PP.timeSinceArrival <= PP_TIME)
```

The first line requires that the first page of the DC jobs is done in maximum FIRST_DC_TIME time units. After this, the PP job may arrive at an unpredictable time. The second and third line help in finding the trade-off between DC job throughput and PP job latency. Variable DC_TIME represents the time distance between the completion of two consecutive pages of DC jobs, whereas variable PP_TIME represents the latency of PP jobs. There is a direct dependency between the time distance between the completion of consecutive DC jobs and their throughput, i.e. a short time distance increases the throughput.

3.5 Results

We found six solutions (strategies), each providing a different trade-off between achieving short time distance between the completion of two consecutive DC jobs and low latency of the PP jobs. We represent these solutions as a *Pareto frontier* in Figure 3.3. A Pareto frontier shows exactly those alternatives that are not dominated by any other alternative. For example, the pairs $(11, 11)$ and $(12, 11)$ are not Pareto optimal because they are dominated by the $(10, 11)$ point. The figure also shows that when the latency is low, the throughput would also be low, and the other way around.

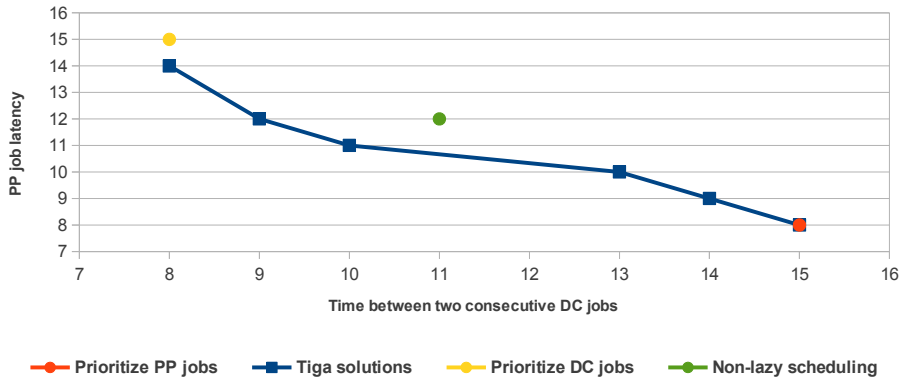


Figure 3.3: The blue line represents the Pareto frontier obtained with UPPAAL-TIGA of the trade-off between the best DC throughput and best PP latency. The dots represent the timings for fixed strategies in which one of the jobs is prioritized or non-lazy scheduling is applied.

UPPAAL-TIGA offers the possibility to export a strategy as a list of rules. However, the multitude of rules (in the order of thousands) provided for each of our solutions impede a direct plug into the controller of an Océ printer. Furthermore, these rules depend on the global state of the system, whereas only part of the global state may be observable by an actual printer controller.

Printer scheduling strategies should be in practice simple and intuitive. This motivates us to check whether we can obtain comparable results as with UPPAAL-TIGA but under settings where we apply some simple scheduling policies suitable for the Océ systems. Consequently, we have come up with three policies: 1) all channels urgent (policy called "non-lazy"), 2) the DC jobs have higher priority when competing for resources against the PP job, and 3) the PP jobs have higher priority under the same circumstances. The three scattered data-points from Figure 3.3 represent the performance of these fixed scheduling policies.

In general, under the "non-lazy" policy, there can be concurrent jobs competing for the same resource, which creates nondeterministic choices that the model checker needs to explore. However, this is not the case for the small number of competing jobs of our scenario. Figure 3.3 also shows that the solution obtained for the "non-lazy" policy is worse than any Pareto point.

By prioritizing the DC jobs, the PP jobs are allowed to use IP2 only when the scanner is in use. This allows us to reach the lowest time distance between consecutive DC jobs but the latency of the PP jobs is higher than any solution obtained with UPPAAL-TIGA.

In the policy that gives priority to the PP jobs, a DC job should not use IP2 if a PP job is downloaded. However, this is not sufficient, since we allow two streams

of DC jobs. It may happen that a DC job is still using the printer when the PP job finishes using IP2 and wants to use the printer. In order to avoid this issue, we have introduced an extra constraint, namely that a DC can use IP2 only if no other DC is using the printer. This implies that never two DC jobs are printed one after another. With the latter constraint, we obtain the same solution as with UPPAAL-TIGA for the lowest PP latency.

3.6 Conclusions and Future Work

We have performed in this chapter trade-off analysis with UPPAAL-TIGA for a common scenario encountered of an Océ datapath where one of the jobs has an uncertain arrival time.

Despite our promising initial results, the problem to automatically synthesize *practical* scheduling strategies for this application domain is still widely open. In some other applications, the strategies synthesized by UPPAAL-TIGA have been used directly in the generation of control software [JRLD07, CJL+09]. However, this is currently (UPPAAL-TIGA, version 0.13) not possible for our printer case study: the strategies produced by UPPAAL-TIGA (albeit memoryless) are really large and contain thousands of rules. Moreover, we had to restrict our analysis to a scenario with a fixed continuous stream of Direct Copy jobs and a single continuous uncontrollable stream of Print with Processing jobs because the version of UPPAAL-TIGA that we have used cannot handle more uncontrollable jobs. Furthermore, the models that we have described in this chapter are simplifications of realistic and more complex models constructed in the context of the Octopus project. We may expect that for the realistic models (e.g. models of Chapter 4), the generated strategies will be far beyond the tight constraints on CPU and memory usage of today's printer controllers.

Nevertheless, we believe that UPPAAL-TIGA can be useful in the actual design of datapath controllers. These controllers typically consist of a relatively small number of simple rules that determine which resource is allocated to which job (e.g. job and resource priorities, FCFS for jobs with equal priority, non-lazy resource allocation). By applying UPPAAL-TIGA to (downsized) versions of printer models and replaying the resulting strategies in the simulator, we may be able to come up with new control rules that are implementable on real printers. UPPAAL-TIGA may also give an indication of how close the implemented rules are from the optimum. Under these assumptions, we can also use the work of Frinkbeiner and Peter [FP12] on controller synthesis based on templates. They give an abstraction-refinement solution to the controller synthesis problem when the size and shape of the controller is fixed in advance by a template. This problem reduces to finding the right actions (encoded as boolean parameters) that would make the controller match the template.

In the next chapter, we refine the model of the Océ printer datapath by including features like memory bus and scheduling rules specific to the Océ controller.

Having defined the controller behavior and being interested in evaluating scenarios with a larger number of jobs (involving for instance batches of several hundred of jobs), we favor UPPAAL for the analysis.

Chapter 4

Dynamic Scheduling with UPPAAL

In multiprocessor systems with many data-intensive tasks, a bus may be among the most critical resources. Typically, high bandwidth allocation to one (high-priority) task may lead to a reduction in the bandwidth of other tasks, which thereby effectively slows down these tasks. Worst-case execution time (WCET) analysis for this type of systems is a major research challenge. In this chapter, we show how the dynamic behavior of a memory bus and a USB in a realistic printer application can be faithfully modeled using timed automata. We analyze, using UPPAAL, the worst case latency of scan jobs with uncertain arrival times in a setting where the printer is concurrently processing an infinite stream of print jobs.

4.1 Introduction

Modern embedded systems are characterized by distributed implementation platforms that include a heterogeneous mix of several processors, one or more buses for communication, and a variety of sensing and actuating devices. They have to operate in dynamic and interactive environments, and need to carry out a mix of data-intensive computational tasks and event-processing control tasks. Not only functional correctness is important, but also quantitative properties related to timeliness, quality-of-service, resource usage, and energy consumption. Nowadays, the complexity of embedded systems and their development trajectories are increasing rapidly. At the same time, development trajectories are expected to deliver products that are inexpensive but meeting stringent time-to-market constraints. The complexity of the designs and the constraints imposed on the development trajectory dictate a systematic, model-driven design approach that leverages reuse

and is supported by tooling whenever possible.

In multiprocessor systems with many data-intensive tasks, a bus may be among the most critical resources, and severely degrade the timing predictability. The problem is that high bandwidth allocation to one (high-priority) task may lead to a reduction of the bandwidth of other tasks, and thereby effectively slowing down these tasks. If we do not want this to occur, for instance in case of safety-critical systems, then we may use e.g. a time division multiple access (TDMA) strategy on the buses in order to give each task a guaranteed bandwidth. However, for most systems such a solution is too expensive. According to Williams et al. [WWP09], for the foreseeable future, off-chip memory bandwidth will often be the constraining resource in system performance of multicore computers. Clearly, WCET analysis for such systems is a major research challenge. Existing performance analysis techniques are not able to accurately predict WCETs for systems with this type of highly dynamic resource behavior. Simulation of detailed models certainly provides insight, but fails to provide WCETs in settings with uncertain job arrival times, dynamic and interactive environments and/or uncertain processing times.

In this chapter, we elaborate on the Océ case study presented in Chapter 2 by adding scheduling rules used by the printer controller. Moreover, we show how to compute WCETs using the model checker UPPAAL [BDL+06, LPY97, BDL04]. To be more specific, we analyze the worst case latency of a job which has uncertain arrival time in a setting where the machine is concurrently processing an infinite stream of other concurrent jobs. In contrast with the previous chapters, the UPPAAL model of this chapter includes all specific scheduling rules that an Océ controller uses to solve resource conflicts. As a result, if job arrival times are known, the UPPAAL model should be deterministic.

The purpose of this chapter is to show that the UPPAAL model checker can handle the complexity of dynamic memory bus behavior in a realistic model of a complex industrial application. To the best of our knowledge, no other analysis techniques/tools, except maybe the hybrid method of [LPT09], is currently able to do an accurate performance analysis for this type of systems (involving a dynamic memory bus and job uncertain arrival times). Existing techniques for WCET analysis of distributed embedded systems, such as Modular Performance Analysis [CKT03, TBHH07], SymTA/S [HJRE04] and MAST [HGGM01] are not applicable since they lead to overly conservative analysis results. In [LPT09], a hybrid method is proposed for analyzing embedded real-time systems that integrates modular performance analysis and timed automata. It would be interesting to use our detailed UPPAAL models of the memory bus and USB as part of this hybrid method.

The structure of this chapter is as follows. The next section introduces the printer case study. In Section 3, the timed automata models are described. The analysis results are expounded in Section 4. Concluding remarks and discussions of future work follow in Section 5.

4.2 Case Study

Figure 4.1 presents the architecture of the Océ datapath analyzed in this chapter. This architecture is a refinement of the datapath architecture presented in Figure 2.1. It contains a memory bus that is the communication medium between resources and memories. High job performance is of critical importance in the Océ datapath design. Therefore, in this chapter we analyze one common scenario: Scan to Email job (Figure 4.2) in parallel with Print from DocBox job (Figure 4.3). We use the term *scan job* for a job which uses the Scan to Email use case and *print job* for a job which uses the Print from DocBox use case. Figures 4.2 and 4.3 also show the dependencies between resources, which are of two types: sequential execution (e.g. IP2 - IP3), or parallel execution (e.g. Upload and Print). The scan jobs use the Scan Memory for temporary storage and print jobs use the Print Memory. These memories limit the number of concurrent jobs in the system.

Océ datapaths employ some specific scheduling rules that solve conflicts which occur among concurrent jobs. These rules greatly reduce the complexity of the control software, and also of the state space. We present here the most important ones, which we have also implemented in our UPPAAL model.

The first rule is *job non-overtaking*: jobs that have the same use case are processed in the order they enter the system.

The second rule is referred to as *bus throttling*. The memory bus is shared among all resources that transfer data to memories. Each resource claims a different percentage of bus bandwidth, but it often occurs that the bandwidth available is not enough. Moreover, the execution time of a resource is limited by the bandwidth it accesses, the internal processing time being negligible. Therefore, good bandwidth management is important for improving system performance. When more than one job is processed, often incoming jobs do not have enough bandwidth available to start. The bus throttling rule is applicable in such conflicting situations. Bus conflicts are solved based on a resource priority list (Figure 4.4). The resources at the top of the list have the highest priority. In general, a resource with high priority receives the bandwidth claimed (Algorithm 1). This potentially reduces the bandwidth of other running resources with lower priority (Lines 3-11), since the total bandwidth used may never exceed an upper bound. Similarly, when a resource finishes a job, it releases the bandwidth occupied (Algorithm 2) which is then further redistributed among the remaining running resources depending on their priority level, the resources with high priority getting more bandwidth back. As mentioned above, the speed of a resource depends directly on the bandwidth assigned to it. Consequently, whenever the bandwidth occupied by a resource is modified, the expected completion time of the corresponding job is also modified. From another perspective, this rule induces a dynamic behavior in the Océ system, which is not easy to predict. Therefore, the bus throttling rule brings a dynamic behavior not only in the USB, but also in the majority of resources.

The third rule is called *upload in order*. The USB client is one of the slowest

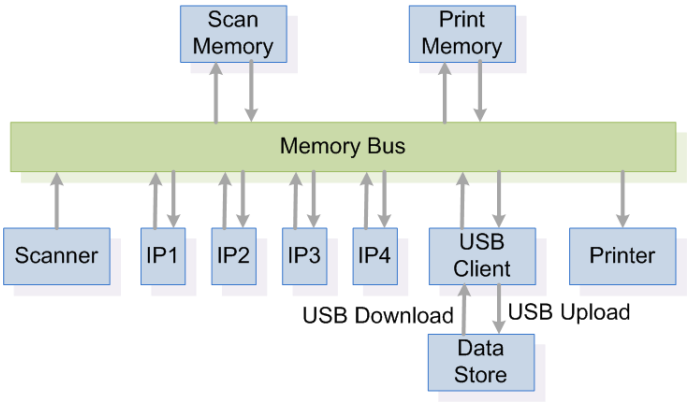


Figure 4.1: Refined architecture of an Océ datapath.

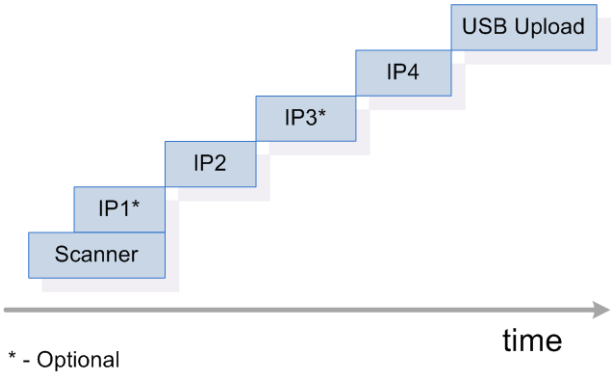


Figure 4.2: Scan to Email use case.

resources in the system. The upload in order rule states that, when a large number of jobs wait for uploading, they should be served in order. For this purpose, a list of waiting jobs is maintained. There is a strict rule when jobs are added to the list. A scan job is inserted after the scanner has completed it, whereas the jobs which use the Print from DocBox use case are added after IP4 has processed them. The same also happens when IP2 is shared, but in this case we add scan jobs after the scanning step and print jobs after the download.

The next rule is called *prioritize print jobs* and refers to conflicts between scan and print jobs for shared resources. Whenever a scan and a print job claim a resource at the same time, the print job gets higher priority.

Finally, we require that all resources are *non-lazy*. This means that if a resource is available when a job claims it, it should immediately start processing the job.

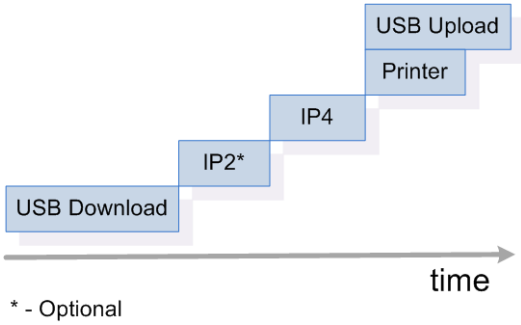


Figure 4.3: Print from DocBox use case.

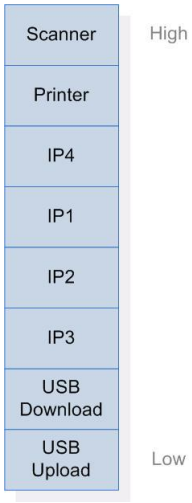


Figure 4.4: Resource priority list.

4.3 Model Description

The timed automata model is structured as follows. Each resource is modeled by an automaton as in Figure 4.5, except the two memories and the memory bus which are simply modeled as shared integer variables. A resource stays in the IDLE location until it is claimed by a job. When a resource is assigned to process a job, the resource computes the bandwidth it can use and the corresponding rate, applying the bus throttling rule when needed and then it jumps into location RUNNING. Then, the resource stays in this location until either the job is completed or its rate is changed. In the latter case, the transition between locations RUNNING and UPDATE_WORK is urgently taken. On the transition between locations UPDATE_WORK and RUNNING, the remaining work is updated in the following way.

Algorithm 1
void allocateBandwidth(resource r)

```
1: if (bw_available<bw_claimed(r)) then
2:    crt_priority = 0;
3:    while (crt_priority < priority(r) && bw_available < bw_claimed(r)) do
4:       r' = resource(crt_priority++);
5:       if bw(r') ≥ (bw_claimed(r)-bw_available) then
6:          bw(r') −= bw_claimed(r) - bw_available;
7:          bw_available = bw_claimed(r);
8:       else
9:          bw_available += bw(r');
10:         bw(r') = 0;
11:      end if
12:   end while
13: end if
14: if (bw_available ≥ bw_claimed(r)) then
15:   bw(r) = bw_claimed(r);
16:   bw_available −= bw_claimed(r);
17: else
18:   bw(r) = bw_available;
19:   bw_available = 0;
20: end if
```

Algorithm 2
void deallocateBandwidth(resource r)

```
1: bw_available += bw(r);
2: bw(r) = 0;
3: crt_priority = priority(r);
4: while (crt_priority > 0 && bw_available > 0) do
5:    r' = resource(crt_priority −−);
6:    if (bw(r') < bw_claimed(r')) then
7:       if (bw_available ≥ (bw_claimed(r')- bw(r'))) then
8:          bw_available −= bw_claimed(r')- bw(r');
9:          bw(r') = bw_claimed(r');
10:      else
11:         bw(r') += bw_available;
12:         bw_available = 0;
13:      end if
14:   end if
15: end while
```

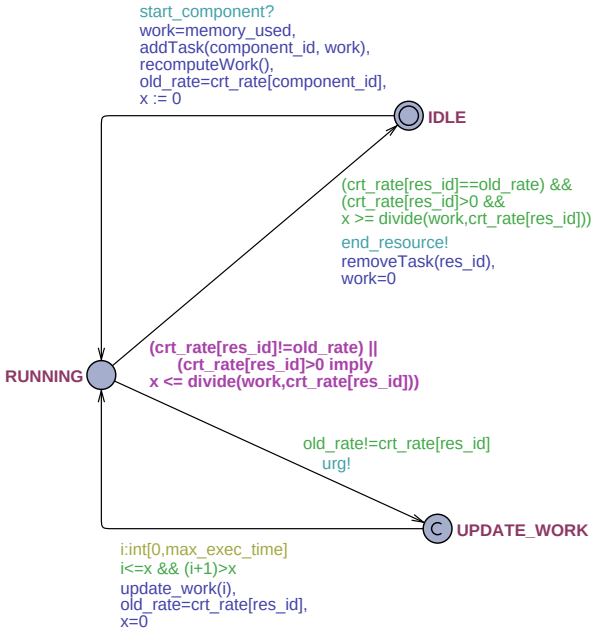


Figure 4.5: Resource automaton.

First, we approximate clock x, which monitors the time elapsed since the latest rate change to the closest integer lower or equal to the clock value (select statement: i:int[0,max_exec_time] and guard: i≤x && i+1>x). Then, the remaining unprocessed data of the current job is updated. All components, except for Printer and Scanner, use this template. Printer and Scanner are never interrupted after they start. Therefore, they do not need the UPDATE_WORK location.

Each job is modeled as a separate automaton. Figure 4.6 displays an automaton representing a simplified version of the Print from DocBox job. For readability reasons, the variables used to model the job non-overtaking rule are not included. The automaton contains actions specific to both the use case (e.g. channels start_down and end_down) and memory management (e.g setPrintMemory(), print_memory += download_memory). In the figure, we can also observe actions that implement the upload in order scheduling rule (e.g. updateUploadOrderArray).

4.4 Verification

In this section we report on the worst case latency of Scan to Email jobs that have uncertain arrival times (modeled using non-determinism by a non-urgent broadcast channel) in a setting where the machine processes in parallel an infinite stream of Print from Docbox jobs. We further assume that each job contains a single-page

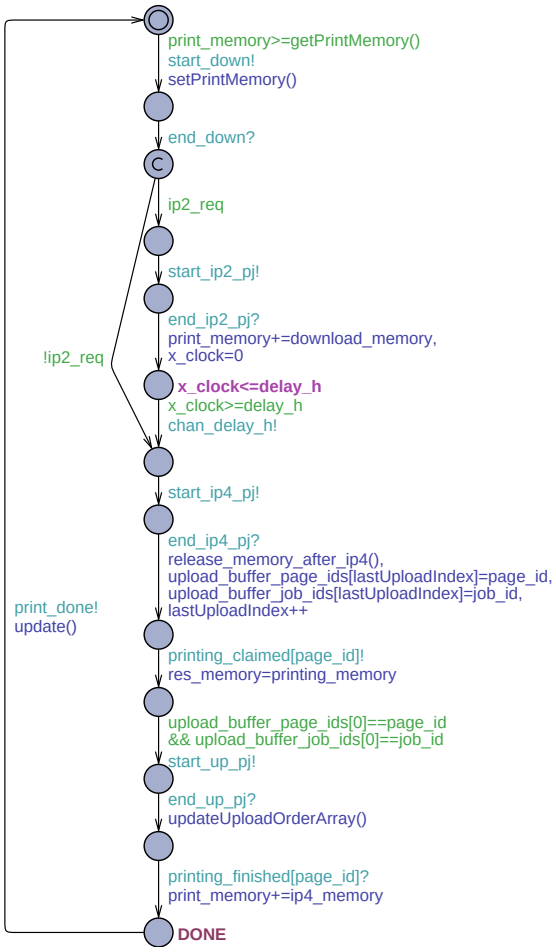


Figure 4.6: Print from DocBox automaton.

file. All experiments were performed using UPPAAL version 4.1.2 on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3GHz) and 128 GB of DDR2 RAM.

Many of the conflicting and nondeterministic situations have been explicitly solved by including the Océ controller scheduling rules in the UPPAAL model. However, the model is still nondeterministic. One cause is the uncertain arrival times of scan jobs. The other cause comes from the multitude of independent actions that may occur simultaneously (e.g. end_down and start_ip2_pj), which UPPAAL explores exhaustively. In order to reduce this, we have specified priorities between all channels and processes of the model. In this way we have predefined an order which UPPAAL chooses to treat simultaneous actions. In addition, when the scan and print jobs accessed shared resources, they used the same channels.

Due to this, we have assigned them separate channels and set higher priority to the channels employed for the communication between print jobs and resources.

The property verified is

```
A[]  ((forall (i:int[0,max_scan_jobs-1])
!ScanJob(i).INIT imply ScanJob(i).latency_clock<=worst_latency)),
```

where we inquire about the worst-case latency of all (max_scan_jobs) scan jobs. The worst-case latency is manually found by repeatedly checking different values for variable worst_latency.

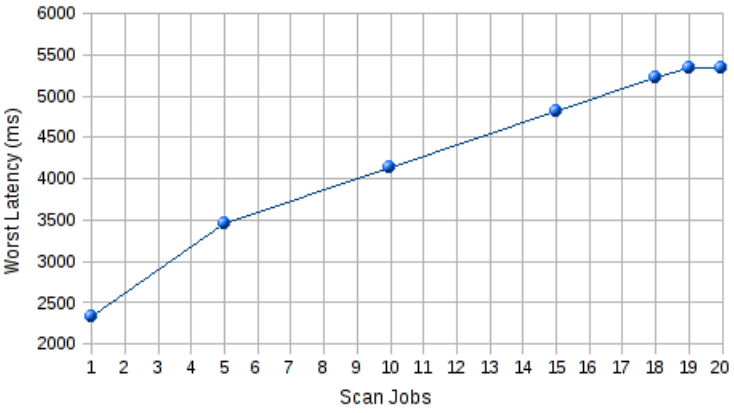


Figure 4.7: Worst scan job latency when no print job is allowed in the system.

Figure 4.7 shows the worst latency as a function of the number of concurrent scan jobs when print jobs were not allowed in the system. We can observe that the maximum scan job load is 19. The last point in the figure indicates that, if the system is fully loaded with scan jobs, the latency of a new job is two times longer than the latency of the first completed scan job. For these experiments the running time of UPPAAL is insignificant. Moreover, the maximum bandwidth occupied is lower than the maximum bandwidth available, therefore, the bus throttling rule has not been applied.

Table 4.1 shows the worst scan job latency in the presence of print jobs. All experiments contained 19 (single-page) scan jobs with uncertain arrival times and a fixed continuous stream of (single-page) print jobs whose size is specified in the first column. The time-out for these experiments is 24 hours. Due to the long running times of UPPAAL, we could not observe the worst scan job latency in combination with high number of print jobs. Twenty print jobs are about forty per cent of the maximum number of infinite print jobs that the system could store. This is, however, expected since the problem we address here is similar to the jobshop scheduling problem, which is notoriously hard (i.e. NP-complete). Nevertheless,

the important observation we can make is that the influence of the print jobs upon the worst case latency of scan jobs is not negligible.

#Print Jobs	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Scan Job Latency(ms)
0	121784	4376.03	723	5341
5	473952	6459.28	349951	6711
10	1107012	15274.40	996693	9843
15	1561524	8934.50	897307	12411
20	-	-	-	-

Table 4.1: Worst scan job latency with print jobs in the system.

During the analysis, we have observed that the bus throttling rule has two negative consequences. On the one hand, this rule worsens scan job latency by slowing the execution time of some resources required by these jobs. On the other hand, it increases the UPPAAL running times due to many changes in the rate of some resources, which UPPAAL has to explore.

#Print Jobs	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Latency(ms)
5	195700	2082.01	147806	4941
10	315540	2232.54	294789	6426
12	384896	2315.72	464711	6767
13	434572	2478.34	555770	6940

Table 4.2: Worst scan job latency of the reduced model.

#Print Jobs	Peak Mem Usage(KB)	Running Time(s)	States Explored	Worst Latency(ms)
5	205684	1940.63	153259	4701
10	304464	2074.39	271351	5933
12	339864	2240.14	331047	6274
13	539340	2539.96	921847	6445

Table 4.3: Worst scan job latency with the improved bus throttling rule.

Because of the great impact of bus throttling on the worst latency of scan jobs, we have further investigated whether there are possible improvements in the resource priority list of this rule. As a consequence, we have reduced the value of some parameters in order to obtain the highest value of the worst scan job latency with UPPAAL. In this sense, the scan memory is reduced by a factor of 2, whereas

the print memory is reduced by a factor of 4. Except for these, nothing else is changed. The scan jobs still have an uncertain arrival time and the print jobs form an infinite stream, but the system hosts fewer jobs. The analysis results of this configuration are detailed in Table 4.2. This model allows maximum 5 scan jobs and 13 print jobs. Furthermore, due to physical constraints only the last four resources in the priority list (see Figure 4.4) can be interrupted after they start processing a job. The optimization found requires to switch the order between USB upload and USB download. With this simple change, we obtain the worst latencies shown in Table 4.3 which are between 4.8% and 7.6% shorter than those of Table 4.2.

Even though we have scaled down the problem, we postulate that the improvement in the priority list applies to any configuration of the datapath analyzed. The reasoning behind this comes from the fact that by partially or completely reducing the bandwidth of the USB download, some jobs are postponed to enter the system, which allows the system to process faster other active jobs. Similarly, by giving more priority to the USB upload, jobs leave the system faster, which would again ease the system load. Furthermore, we could have reduced the priority of resources above the USB in the priority list, but, as mentioned above, some of resources like Scanner or Printer cannot be physically interrupted after they start. In addition, a reduction of the priorities of IP2-IP4 resources would have improved only the performance of the jobs where they are required. Finally, a prolongation of the execution of IP1, which is relatively early in the list of resources claimed by jobs and, moreover, often shared between jobs, would have degrade job latency.

4.5 Conclusions

In this chapter, we have presented the most important scheduling rules that characterize the behavior of an Océ printer controller. In addition, we have computed the worst latency of one job with uncertain arrival time in a setting where another job is used infinitely often. Our results show a strong dependency between the two jobs.

As usual with model checking, long running times were a key issue within our case study. In order to improve one scheduling rule and obtain analysis results faster, we had to scale down some of the parameters in the model. Another technical issue we have faced is that although essentially the behavior of the model is fully deterministic (due to the scheduling rules we added) for each job once it enters the system, the resulting UPPAAL model is not fully deterministic (and suffers from state space explosion) due to the interleaving of internal actions of various resources. We resolved this by assigning priorities to channels and processes defined in the UPPAAL model. As future work, it would be interesting to remodel this case study with UPPAAL-PORT [Uppa] (an extension of UPPAAL that supports partial order reduction) and see whether the analysis is performed faster for the scenario analyzed in this chapter.

We computed the worst latency by repeatedly checking an invariant property. Using a binary search we managed to find the exact value of certain parameters. However, this type of parametric analysis requires a lot of time and it would be more helpful to automate it in UPPAAL, possibly using multiple processors to parallelize computations.

The lesson that we have learned from this case study is that it is extremely difficult to maintain correctness of the model in a setting where the object of modeling has such a high complexity. There was not a single document describing the entire design. In fact there was not a single person who was able to answer all our questions; the knowledge was spread over a large design team. Furthermore, it is difficult for engineers to understand the intricacies of our UPPAAL model. The syntax of UPPAAL is not sufficiently expressive to describe the design in such a way that a small change in the design corresponds to a small change in the model. Due to these difficulties, the Octopus project has decided to develop a high-level language for describing datapath designs, together with a translation to UPPAAL: on the one hand this will ease the communication with the Océ design engineers, and on the other hand it will reduce the chances of introducing errors in the UPPAAL model. The next chapter defines some key constraints from this new language and proves the correctness of the UPPAAL translation.

Chapter 5

Real Time Task Systems

Inspired by work on model-based design of printers, the notion of a parametrized partial order (PPO) was introduced recently. PPOs are a simple extension of partial orders, expressive enough to compactly represent large task graphs with finite repetitive behavior. We present a translation of the PPO subclass to timed automata and prove that the transition system induced by the UPPAAL models is isomorphic to the configuration structure of the original PPO. Moreover, we introduce real-time task systems (RTTSs), a general model for real-time embedded systems that we use to describe the datapath design described in Chapter 2. In an RTTS, tasks are represented as PPOs and the pace of a task instance may vary, depending on the resources that are allocated to it. We describe a translation of a subclass of RTTSs to UPPAAL, and establish, for an even smaller subclass, bisimulation equivalence between the timed configuration semantics of an RTTS and the transition system induced by the corresponding UPPAAL translation. Lastly, we report on a series of experiments which demonstrates that the resulting UPPAAL models are more tractable than handcrafted models of the same systems used in earlier case studies.

5.1 Introduction

Many methods and tools for real-time embedded systems design implement the Y-chart pattern [BCG+97, KDVvdW97]. In the Y-chart pattern, applications, platforms and mappings of applications onto the platforms are described as separate modules. This allows independent evaluation of various alternatives of one of these modules while fixing the others. Moreover, this pattern facilitates the reuse of modules for the design of multiple variants on the same product. Diagnostic information can also be used to, automatically or manually, improve these modules.

Applications are typically described in terms of task graphs representing partially ordered sets of tasks. In practice, we frequently see that certain tasks need to be executed repetitively, for a finite number of times, and that there exists a hierarchical relationship between tasks. For instance, a manufacturing order of a beer brewery consists of several pallets, containing several crates, each containing several bottles of beer. Another example concerns a wafer scanner manufacturing system from the semiconductor industry. Wafers are produced in batches (lots). A wafer scanner projects a mask on a wafer, using light. Eventually, the projected masks result in integrated circuits (ICs). On one wafer, multiple ICs and types of ICs are manufactured. Multiple types of ICs involve multiple masks, and multiple masks are placed on a reticle. As a final example, we mention a copier machine, which has to process a certain number of copies of a file, which in turn consists of a certain number of pages. Due to the nested, finite repetitive behavior, task graphs tend to become very large and no longer practical for specification and analysis of application behavior. Following [NMFR03, HvdNV03], we argue that repetitive task structure of applications plays an important role in real-time embedded systems design, and needs to be addressed in methods for specifying and reasoning about such systems. Repetitive execution of tasks leads to finite repetitive patterns in schedules. In practice, execution of the first few instances and last few instances of a task differs slightly from the rest. This is a large difference with unlimited repetitive ('periodic') behavior, which has received much attention in the scheduling literature.

Within concurrency theory, several semantic models have been proposed that are based on partial ordering of events such as Mazurkiewicz [Maz88] traces, pomsets (partially-ordered multisets) [Pra86], and event structures [Win89], but these models do not incorporate an explicit notion of repetitive events. Partial orderings of events with repetition can be defined using Colored Petri Nets [JKW07, JK09], but this is an extremely rich and expressive formalism, which may be considered too complicated for the task at hand.

The Octopus project has developed a Design-Space Exploration (DSE) toolset [Oct11] that aims to leverage existing modeling, analysis, and DSE tools to support model-driven DSE for real-time embedded systems [BVBG+10]. The Octopus toolset is centered on a Y-chart based intermediate representation, DSEIR (Design-Space Exploration Intermediate Representation), to capture design alternatives. DSEIR models can be exported to various analysis tools. This facilitates the reuse of models across tools and provides model consistency between analyses. The use of an intermediate representation also supports domain-specific abstractions and reuse of tools across application domains. Octopus DSE toolset integrates CPN Tools [JKW07, JK09] for stochastic simulation of timed systems, SDF3 [SGB06] for worst-case throughput calculation, and UPPAAL [BDL04] for model checking and schedule optimization. The initial version[1] of DSEIR is based on the notion of the notion of parametrized partial orders [TVB11]. PPOs are a simple extension of

[1] The latest version of DSEIR has an operational application model [THB+11].

partial orders, but expressive enough to compactly represent large task graphs with repetitive behavior. In DSEIR 1.0, applications are represented as PPOs. This intermediate representation can be translated into the input formats of CPN Tools and UPPAAL. A translation of PPOs to CPN Tools has recently been described in [TVB11]. In this chapter, we define a restricted version of PPOs that is more amenable to model checking. Moreover, we give a translation into timed automata, the semantic model underlying UPPAAL.

We have seen in previous chapters that UPPAAL is able to handle industrial sized designs, but the tool is really pushed to its limits. Therefore, it is crucial to have a translation from PPOs to UPPAAL that is maximally efficient. By unfolding a PPO into a task graph and introducing a separate automaton for each task in the unfolding, we obtain a general translation of PPOs to UPPAAL. However, especially when we have many repetitive events (e.g. print a 300-page document) the translation becomes intractable. Based on the observation that in practice the PPOs often contain tasks that are not auto-concurrent and precedence relations between task instances obey certain monotonicity conditions, we define a subclass of PPOs that allows a more efficient translation.

In the literature, numerous modeling frameworks for real-time task systems have been proposed, see for instance [LL73, MFC01, CKT03, HHJ+05, NWY99, KSY07, FKPY07]. In the design space exploration of printers, a small gain (5%) of performance can be decisive, and consequently detailed models of real-time task systems are needed in which, for instance, we can express that the completion time of a task slightly increases if another task is using the same communication bus simultaneously. To the best of our knowledge, such a refined modeling is not possible in existing frameworks for task systems. Therefore, we introduce, in this chapter, real-time task systems (RTTS), a general model for real-time embedded systems able to accurately describe the datapath of realistic Océ systems. In an RTTS, tasks are represented as PPOs and the pace of a task instance may vary, depending on the resources that are allocated to it. This allows us to accurately model common scenarios in which, for instance, the completion time of a task depends on the varying amount of memory and communication bandwidth allocated to it over time.

This brings us to the main contributions of this chapter:

1. A definition of a PPO subclass and its translation to UPPAAL together with a correctness proof (the transition system induced by the UPPAAL model is isomorphic to the configuration structure of the PPO).

2. A notion of a real-time task system (RTTS), a general model for real-time embedded systems that we use to describe the datapaths of realistic printer designs. We describe a translation of a subclass of RTTSs to UPPAAL, and establish, for an even smaller subclass, bisimulation equivalence between the timed configuration semantics of an RTTS and the transition system induced by the corresponding UPPAAL translation.

3. A series of experiments which demonstrate that UPPAAL models obtained through this translation are more tractable than the handcrafted models of Chapter 2.

The structure of this chapter is as follows. Section 5.2 recalls some preliminary definitions regarding labeled transition systems, the underlying semantic notion used throughout the chapter. Section 5.3 defines PPOs and their semantics, and the translation of a subset of PPOs into networks of timed automata together with a proof of its correctness. Section 5.4 introduces the new model of RTTS. Section 5.5 explains how the translation of PPOs to UPPAAL can be lifted to RTTSs. Section 5.6 presents performance evaluation results of models generated by comparing them with handcrafted UPPAAL models of Chapter 2. Concluding remarks and future work follow in Section 5.7. The models generated for Section 5.6 are available at `www.mbsd.cs.ru.nl/publications/papers/fvaan/HIV12`.

5.2 Preliminaries

We use $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{>0}$ to denote the sets of nonnegative and positive real numbers, respectively, and $\mathbb{N}$ to denote the set of natural numbers.

If X and Y are sets then we write $X \hookrightarrow Y$ for the set of partial functions from X to Y. Given a partial function $f \in X \hookrightarrow Y$, we write $f(x)\downarrow$ if $f(x)$ is defined, and $f(x)\uparrow$ if $f(x)$ is undefined, for $x \in X$.

A *labeled transition system (LTS)* is a tuple $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$, where

- S is a set of states,
- $s_0 \in S$ is an initial state,
- Σ is a set of action labels, and
- $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation.

We write $s \xrightarrow{a} s'$ iff $(s, a, s') \in \rightarrow$ and $s \rightarrow s'$ if there exists an action $a \in \Sigma$ such that $s \xrightarrow{a} s'$. A *path* of $\mathcal{L}$ is a sequence of states $\pi = s_0 s_1 \cdots s_n$ such that, for all $0 \leq i < n$, $s_i \rightarrow s_{i+1}$. In this case we say that π is a path from s_0 to s_n. A state $s \in S$ is *reachable* in $\mathcal{L}$ if there exists a path from s_0 to s. We say that $\mathcal{L}$ is *deterministic* if, for each state $s \in S$ and for each action label $a \in \Sigma$, $s \xrightarrow{a} s'$ and $s \xrightarrow{a} s''$ implies $s' = s''$.

Two labeled transition systems $\mathcal{L}_1 = (S_1, s_0^1, \Sigma_1, \rightarrow_1)$ and $\mathcal{L}_2 = (S_2, s_0^2, \Sigma_2, \rightarrow_2)$ are *isomorphic* if $\Sigma_1 = \Sigma_2$ and there exists a bijective function $f : S_1 \rightarrow S_2$ such that:

- $f(s_0^1) = s_0^2$ and
- $s \xrightarrow{a}_1 s' \Leftrightarrow f(s) \xrightarrow{a}_2 f(s')$, for all $s, s' \in S_1$, $a \in \Sigma_1$.

We say that $\mathcal{L}_1$ and $\mathcal{L}_2$ are *bisimilar* if $\Sigma_1 = \Sigma_2$ and there exists a relation $R \subseteq S_1 \times S_2$ such that $(s_0^1, s_0^2) \in R$ and, for each $(s, r) \in R$,

- if $s \xrightarrow{a}_1 s'$ then there exists a state $r' \in S_2$ such that $r \xrightarrow{a}_2 r'$ and $(s', r') \in R$, and
- if $r \xrightarrow{a}_2 r'$ then there exists a state $s' \in S_1$ such that $s \xrightarrow{a}_1 s'$ and $(s', r') \in R$.

Given an LTS $\mathcal{L} = (S, s_0, \Sigma, \rightarrow)$, we define $\mathsf{reach}(\mathcal{L})$ as the LTS $(S', s_0, \Sigma, \rightarrow')$, where S' is the set of reachable states of $\mathcal{L}$ and $\rightarrow' = \{(s, a, s') \mid s, s' \in S' \wedge s \xrightarrow{a} s'\}$.

5.3 Parameterized Partial Orders

A *parametrized partial order (PPO)* is a partial order that comes equipped with some extra structure to capture repetitive behavior. In [TVB11], a PPO is defined at task level and assumes a precedence relation between tasks. Here, we view a PPO from a different angle where tasks are decomposed into events and a PPO imposes a partial order relation at event level. This perspective allows us to introduce a subclass of PPOs that can be efficiently translated into networks of automata, and later in this section we establish the correctness of this translation.

5.3.1 Definition of PPOs

Tasks in a PPO may be executed repeatedly: each task has a collection of parameters and each valuation of these parameters defines a task instance. The events in a PPO are structured and correspond to either the start or the end of a task instance.

Formally, we assume a universe $\mathcal{P}$ of typed variables called *parameters*. A *valuation* of a set $P \subseteq \mathcal{P}$ of parameters is a function that maps each parameter in P to an element of its domain. We assume that the domain of each parameter is a nonempty set. We write $V(P)$ for the set of valuations of variables in P.

A *parameterized partial order (PPO)* is a tuple $\mathcal{A} = (\mathcal{T}, \mathcal{M}, E, U)$ where

- $\mathcal{T}$ is a finite set of *tasks*. We define the set of *event types* by $\mathcal{E} = \{s, e\} \times \mathcal{T}$. Projection functions $\mathsf{task} : \mathcal{E} \rightarrow \mathcal{T}$ and $\mathsf{type} : \mathcal{E} \rightarrow \{\mathsf{s}, \mathsf{e}\}$ are given by $\mathsf{task}((t, T)) = T$ and $\mathsf{type}((t, T)) = t$, and embeddings $\mathsf{start} : \mathcal{T} \rightarrow \mathcal{E}$ and $\mathsf{end} : \mathcal{T} \rightarrow \mathcal{E}$ are given by $\mathsf{start}(T) = (s, T)$ and $\mathsf{end}(T) = (e, T)$, with $t \in \{s, e\}$, and $T \in \mathcal{T}$.
- $\mathcal{M}$ is a function that assigns to each task T a finite set of parameters in $\mathcal{P}$; we write $V(T)$ as a shorthand for $V(\mathcal{M}(T))$.
- $E \subseteq \mathcal{E} \times \mathcal{E}$ is a set of *edges*. We require, for each $T \in \mathcal{T}$, $(\mathsf{start}(T), \mathsf{end}(T)) \in E$.

- For each edge $p = (A, B) \in E$, $U(p) : V(\mathsf{task}(A)) \hookrightarrow V(\mathsf{task}(B))$ is a *precedence function*. We write $A \xrightarrow{u} B$ if $(A, B) \in E$ and $U(A, B) = u$. We require that the start of a task instance precedes the end of that instance, i.e., for each task $T \in \mathcal{T}$ and valuation $v \in V(T)$, $U((\mathsf{start}(T), \mathsf{end}(T)))(v) = v$.

Below, we present two examples that illustrate how PPOs can be used to model scheduling applications.

Printer Figure 5.1a depicts a part of an application encountered in the printer domain (see [IKY+08]). There are three tasks: Scan, ScanIP and Delay, represented by rectangles. The corresponding start and end event types are indicated by subrectangles inscribed with s and e. Edges show the dependencies between event types (the edges from start to corresponding end are not shown). All three tasks have one parameter: p of type $[0, \ldots, L]$ representing the number of the current page processed. The constant $L \in \mathbb{N}$ is a bound for the parameter p. A precedence function $A \xrightarrow{u} B$ is represented by a predicate that may contain both the parameters of $\mathsf{task}(A)$ and primed versions of the parameters of $\mathsf{task}(B)$. For instance, the predicate $\mathsf{p}' = \mathsf{p} + 1$ on the edge from ScanIP to Scan represents the precedence function that maps a valuation v of the ScanIP parameters to the unique valuation v' of the Scan parameters that satisfies $v'(\mathsf{p}) = v(\mathsf{p}) + 1$.

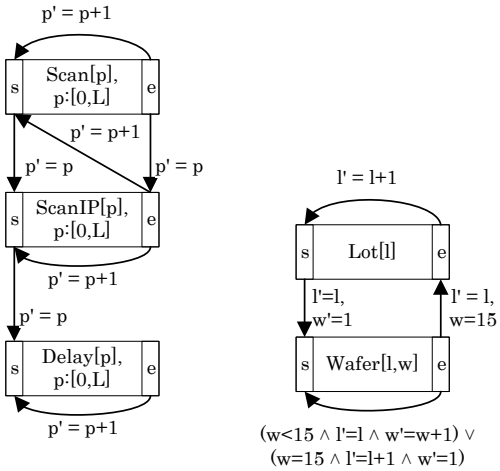
An instance of ScanIP may start as soon as its corresponding instance of Scan has started. These task instances of Scan and ScanIP may then proceed in parallel. However, the next instance of Scan may only start after the current instances of both Scan and ScanIP have ended. Between the ScanIP and Delay tasks, there is also a dependency: only after the occurrence of the start event in the ScanIP task, the start event of the corresponding Delay task may occur.

Wafer production The PPO displayed in Figure 5.1b describes the production of an infinite series of lots, where each lot is composed of 15 wafers. This example is inspired by [HvdNV03]. After the start of each lot, 15 wafer tasks are executed in sequence, followed by the end of the lot.

5.3.2 From PPOs to Configuration Structures

The semantics of a PPO can be described in terms of a labeled transition system, referred to as the *configuration structure* of the PPO (see [Win89, vGP09]). The states of a configuration structure are *configurations*, finite sets of events that have already occurred. Each transition marks the occurrence of a single new event for which all the immediate predecessors have occurred.

Formally, an *event* is a pair (A, v) where A is an event type and $v \in V(\mathsf{task}(A))$ is a valuation of its task parameters. We write $\mathsf{ev\_type}((A, v)) = A$ and $\mathsf{task}((A, v)) = \mathsf{task}(A)$. Also, we write $\mathsf{ev}(\mathcal{A})$ for the set of events of a PPO $\mathcal{A}$. We call event (B, w) an *immediate predecessor* of event (A, v), notation $(B, w) \mapsto (A, v)$, if $(B, A) \in E \wedge U(B, A)(w) = v$.



(a) Printer (b) Wafer production

Figure 5.1: PPO representation.

Let $C \subset \mathsf{ev}(\mathcal{A})$ and $\alpha \in \mathsf{ev}(\mathcal{A})$ with $\alpha \notin C$. We say that C *enables* α, and write $C \vdash \alpha$, if all immediate predecessors of α are in C.

Let $\mathcal{A}$ be a PPO. The set $\mathsf{conf}(\mathcal{A})$ of *configurations* of $\mathcal{A}$ is the smallest subset of the power set $\wp(\mathsf{ev}(\mathcal{A}))$ of events of $\mathcal{A}$ such that:

1. $\emptyset \in \mathsf{conf}(\mathcal{A})$,
2. if $C \in \mathsf{conf}(\mathcal{A})$, and $C \vdash \alpha$ then $C \cup \{\alpha\} \in \mathsf{conf}(\mathcal{A})$.

The *configuration structure* of $\mathcal{A}$ is the LTS $\mathcal{C}(\mathcal{A}) = (\mathsf{conf}(\mathcal{A}), \emptyset, \mathcal{E}, \rightsquigarrow)$, where $(C, A, C') \in \rightsquigarrow$ iff $C \in \mathsf{conf}(\mathcal{A})$ and there exists an $\alpha \in \mathsf{ev}(\mathcal{A})$ s.t. $C \vdash \alpha$, $\mathsf{ev\_type}(\alpha) = A$ and $C' = C \cup \{\alpha\}$. We write $C \overset{A}{\rightsquigarrow} C'$ if $(C, A, C') \in \rightsquigarrow$. Also, we sometimes write $C \overset{\alpha}{\rightsquigarrow} C'$ for $C \overset{\mathsf{ev\_type}(\alpha)}{\rightsquigarrow} C'$ and $C' = C \cup \{\alpha\}$.

The above definition implies that each configuration $C \in \mathsf{conf}(\mathcal{A})$ has a *securing*, i.e., a sequence $\alpha_1, \ldots, \alpha_n$ of events such that $C = \{\alpha_1, \ldots, \alpha_n\}$ and, for each $1 \leq i \leq n$, $\{\alpha_j \mid j < i\} \in \mathsf{conf}(A)$ and $\{\alpha_j \mid j < i\} \vdash \alpha_i$.

In a PPO, there are no conflicts between events: it is not possible that the occurrence of one event disables the occurrence of another event. In fact, it is easy to prove that the set of configurations of a PPO is closed under union: if $C \in \mathsf{conf}(\mathcal{A})$ and $C' \in \mathsf{conf}(\mathcal{A})$ then $C \cup C' \in \mathsf{conf}(\mathcal{A})$. We call an event *reachable* if it occurs in some configuration, and write $\mathsf{rev}(\mathcal{A})$ for the set of reachable events of $\mathcal{A}$. Note that, since in a PPO we allow cyclic predecessor relations, it may occur that some (or even all) events are not reachable. If α and α' are in $\mathsf{rev}(\mathcal{A})$, we write $\alpha \leq_{\mathcal{A}} \alpha'$, if for each configuration $C \in \mathsf{conf}(\mathcal{A})$, $\alpha' \in C$ implies $\alpha \in C$. The technical lemma below states that the $\leq_{\mathcal{A}}$ contains the immediate predecessor relation:

Lemma 5.3.1 *Let $\mathcal{A}$ be a PPO with events α and α' such that $\alpha \mapsto \alpha'$. Then $\alpha' \in \mathsf{rev}(\mathcal{A})$ implies $\alpha \in \mathsf{rev}(\mathcal{A})$ and $\alpha \leq_{\mathcal{A}} \alpha'$.*

Proof If $\alpha' \in \mathsf{rev}(\mathcal{A})$, then there is a configuration $C \in \mathsf{conf}(\mathcal{A})$ that contains α'. Furthermore, this configuration has a securing, i.e., a sequence $\alpha_1, ..., \alpha_n$ such that $C = \{\alpha_1, ..., \alpha_n\}$ and there is a configuration C'_α that we can construct with some of the events of C that enables α'. Since $C'_\alpha \vdash \alpha'$, C'_α contains all immediate predecessors of α'. Let α be an immediate predecessor of α'. Since $\alpha \in C'_\alpha$, then $\alpha \in \mathsf{rev}(\mathcal{A})$. Because $C'_\alpha \subset C$, then $\alpha \in C$, namely α is included in any configuration that contains α', therefore, $\alpha \leq_{\mathcal{A}} \alpha'$.

The following lemma states that a PPO induces a partial ordering relation on its (reachable) events.

Lemma 5.3.2 *Let $\mathcal{A}$ be a PPO, then $\leq_{\mathcal{A}}$ is a partial order on $\mathsf{rev}(\mathcal{A})$.*

Proof

1. (Reflexivity). We need to prove that $\alpha \leq_{\mathcal{A}} \alpha$ is true, for any $\alpha \in \mathsf{rev}(\mathcal{A})$, which holds.

2. (Antisymmetry). Let $\alpha, \alpha' \in \mathsf{rev}(\mathcal{A})$. We should prove that if $\alpha \leq_{\mathcal{A}} \alpha'$, and $\alpha' \leq_{\mathcal{A}} \alpha \implies \alpha = \alpha'$. Assume that $\alpha \neq \alpha'$. If $\alpha \leq_{\mathcal{A}} \alpha'$, there exists a securing whose configuration C'_α contains α and enables α'. Further, if $\alpha' \leq_{\mathcal{A}} \alpha$, there exists a securing whose configuration C_α contains α' and enables α and $C'_\alpha \subset C_\alpha$. This implies that $\alpha \in C_\alpha$ and $C_\alpha \vdash \alpha$, which is impossible.

3. (Transitivity). Let $\alpha, \alpha', \gamma \in \mathsf{rev}(\mathcal{A})$. We should prove that if $\alpha \leq_{\mathcal{A}} \alpha'$, and $\alpha' \leq_{\mathcal{A}} \gamma \implies \alpha \leq_{\mathcal{A}} \gamma$. Assuming that $\alpha \leq_{\mathcal{A}} \alpha'$, this implies that any configuration that contains α' also contains α. Further, if $\alpha' \leq_{\mathcal{A}} \gamma$, any configuration that contains γ also contains α'. Since any configuration that contains α', also contains α, this allows us to conclude that any configuration that contains γ also contains α, and therefore, $\alpha \leq_{\mathcal{A}} \gamma$.

5.3.3 Restricted PPOs

We explore the behavior of PPOs using the UPPAAL model checker, and for this we need to translate PPOs to the input language of UPPAAL. Here we describe a translation of a subclass of PPOs in which no two instances of a task can run concurrently. It is possible to translate arbitrary PPOs to UPPAAL (provided the parameter domains are finite) but this translation leads to networks of automata that are much harder to analyze. Moreover, the Océ datapath design problem studied in this thesis satisfies this restriction.

We call a PPO $\mathcal{A}$ *restricted* if it satisfies the following five conditions, for all tasks T and T', for all precedence functions $A \xrightarrow{u} B$ with $\mathsf{task}(A) = T$ and $\mathsf{task}(B) = T'$, and for all valuations $v, w \in V(T)$:

- **C0**: The only edges between events of the same task are the ones from the start event to the end event, and from the end event to the start event:

$$\mathsf{task}(A) = \mathsf{task}(B) \quad \Rightarrow \quad ((A, B) \in E \Leftrightarrow A \neq B)$$

  We write $\mathsf{next}(T)$ for the function $U((\mathsf{end}(T), \mathsf{start}(T))$, and let $<_T$ be the least transitive relation on valuations in $V(T)$ satisfying $v <_T \mathsf{next}(T)(v)$. Write $v \leq_T w$ iff $v <_T w$ or $v = w$.

- **C1**: There is exactly one valuation of the parameters of T that does not appear in the range of $\mathsf{next}(T)$. This valuation is referred to as the *initial valuation* of T, and is written v^0_T.

- **C2**: $\mathsf{next}(T)$ is injective

- **C3**: u is only defined for reachable valuations:

$$u(v)\downarrow \quad \Rightarrow \quad v^0_T \leq_T v$$

- **C4**: u is *monotonic*:

$$v \leq_T w \wedge u(w)\downarrow \quad \Rightarrow \quad u(v)\downarrow \wedge\, u(v) \leq_{T'} u(w)$$

Axioms **C0**, **C1** and **C2** impose precedence restrictions between event instances of the same task that exclude auto-concurrency. Axiom **C0** implies that we have an edge from the end event type of a task to the corresponding start event type. Axiom **C1** implies that, for each task, there is only one event that does not depend on some other event of the same task: necessarily this is going to be the first event of the task that will occur. Axiom **C2** implies that each event of a task, except the initial one, has a unique immediate predecessor event that belongs to the same task. Axioms **C0-C2** still allow cyclic precedence edges between events of the same task, but axiom **C3** implies that u is not defined for such "ghost events". Axiom **C4**, finally, states that a precedence function that links events of different tasks is monotonic w.r.t the event ordering within tasks. The reader may check that the examples of Section 5.3.1 are restricted.

Lemma 5.3.3 *Let $\mathcal{A}$ be a restricted PPO. Given a task T and valuation v. Then*

1. $(\mathsf{end}(T), v) \in \mathsf{rev}(\mathcal{A})$ *implies* $(\mathsf{start}(T), v) \in \mathsf{rev}(\mathcal{A})$ *and* $(\mathsf{start}(T), v) \leq_{\mathcal{A}} (\mathsf{end}(T), v)$.

2. $(\mathsf{start}(T), \mathsf{next}(T)(v)) \in \mathsf{rev}(\mathcal{A})$ *implies* $(\mathsf{end}(T), v) \in \mathsf{rev}(\mathcal{A})$ *and* $(\mathsf{end}(T), v) \leq_{\mathcal{A}} (\mathsf{start}(T), \mathsf{next}(T)(v))$.

3. $\leq_{\mathcal{A}}$ *is a total order on the set* $\{\alpha \in \mathsf{rev}(\mathcal{A}) \mid \mathsf{task}(\alpha) = T\}$ *of reachable events of* T.

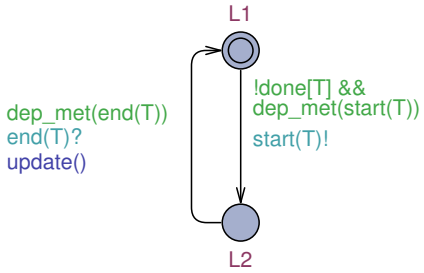


Figure 5.2: UPPAAL automaton for task T.

Proof Statements (1) and (2) follow by Lemma 5.3.1.
For (3), first observe that $\leq_{\mathcal{A}}$ is a partial order on $\mathsf{rev}(\mathcal{A})$ by Lemma 5.3.2. Hence it is also a partial order on the subset of reachable events of T. Let $\alpha, \alpha' \in \mathsf{rev}(\mathcal{A})$ with $\mathsf{task}(\alpha) = T$ and $\mathsf{task}(\alpha') = T$. It suffices to prove that either $\alpha \leq_{\mathcal{A}} \alpha'$ or $\alpha' \leq_{\mathcal{A}} \alpha$. Assuming that $\alpha = (t_\alpha, v_\alpha), \alpha' = (t'_\alpha, v'_\alpha)$, with $t_\alpha, t'_\alpha \in \{\mathsf{end}(T), \mathsf{start}(T)\}$. If $v_\alpha = v'_\alpha$, then by applying the first case of this lemma, it follows that $\alpha \leq_{\mathcal{A}} \alpha'$ or $\alpha' \leq_{\mathcal{A}} \alpha$. If not, without loss of generality, we assume $v_\alpha <_T v'_\alpha$. Then using the first two cases of this lemma, it follows that $(t_\alpha, v_\alpha) \leq_{\mathcal{A}} (\mathsf{start}(T), \mathsf{next}(T)(v_\alpha)) \leq_{\mathcal{A}} \ldots \leq_{\mathcal{A}} (t'_\alpha, v'_\alpha)$, which by transitivity implies that $\alpha \leq_{\mathcal{A}} \alpha'$.

5.3.4 From restricted PPOs to networks of automata

We will show how each restricted PPO can be translated into a UPPAAL network of automata in such a way that (the reachable part of) the LTS induced by the composition of these automata is isomorphic to the configuration structure of the PPO.

Let $\mathcal{A}$ be a restricted PPO. We define $\mathcal{N}(\mathcal{A})$ to be the LTS induced by the network of automata obtained by instantiating the template displayed in Figure 5.2, for each task $T \in \mathcal{T}$ (cf. Definition 3 of [BDL04]). Below we explain the various predicates and functions occurring in Figure 5.2. The composed system $\mathcal{N}(\mathcal{A})$ has the following set of global shared variables:

$$\{T.\mathsf{p}, \mathsf{loc}[T], \mathsf{done}[T] \mid T \in \mathcal{T} \wedge \mathsf{p} \in \mathcal{M}(T)\}.$$

Variable $\mathsf{loc}[T]$ records the current location of the task automaton for T, which can be either $\mathsf{L1}$ or $\mathsf{L2}$. Boolean variable $\mathsf{done}[T]$ records whether the last event of T has been executed. Since different tasks may use the same parameter names, we make a copy $T.\mathsf{p}$ of each parameter $\mathsf{p} \in \mathcal{M}(T)$. As long as task T has not yet been completed, variable $T.\mathsf{p}$ gives the value of p in the next event of T that will occur. Variable $\mathsf{loc}[T]$ is initialized to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is initialized to false, and variable $T.\mathsf{p}$ is initialized to $v_T^0(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

For a given state of the automaton for task T, let function $\mathsf{val}(T)$ return the current valuation of the parameters of task T. For each event type A with $\mathsf{task}(A) = T$, function $\mathsf{done}(A)$ returns true iff the last event of A has occurred:

$$\begin{aligned}\mathsf{done}(A) \quad = \quad & \mathsf{done}[T] \vee (\mathsf{loc}[T] = \mathsf{L2} \wedge \\ & \mathsf{type}(A) = \mathsf{s} \wedge \mathsf{next}(T)(\mathsf{val}(T)) \uparrow)\end{aligned}$$

If the last event of A has not occurred, function $\mathsf{next}(A)$ gives the valuation of the parameters for the next event of A:

$$\mathsf{next}(A) = \begin{cases} \mathsf{next}(T)(\mathsf{val}(T)), \text{if } \mathsf{loc}[T] = \mathsf{L2} \wedge \mathsf{type}(A) = \mathsf{s} \\ \mathsf{val}(T) \qquad\qquad\quad , \text{otherwise} \end{cases}$$

Suppose that the last event of type A has not occurred, then in order to decide whether the next event of A may occur, we check for each incoming precedence edge $B \xrightarrow{u} A$ whether the dependency induced by that edge has been met:

$$\begin{aligned}\mathsf{dep\_met}(A) \quad = \quad & \forall B, u : B \xrightarrow{u} A \wedge \mathsf{task}(B) \neq \mathsf{task}(A) \implies \\ & \mathsf{dep\_met}(B, u, A)\end{aligned}$$

Note that the task automaton already takes care of the dependencies induced by precedence functions between pairs of start and end events of T. To decide whether the dependencies induced by $B \xrightarrow{u} A$ are met, we first check if $\mathsf{done}(B)$ evaluates to true. If so then all events of B have occurred and hence all dependencies induced by $B \xrightarrow{u} A$ have been met. Next we check whether $u(\mathsf{next}(B))$ is defined. If not then, by monotonicity, all dependencies induced by $B \xrightarrow{u} A$ have been met. Finally, we check whether $\mathsf{next}(A)$ precedes $u(\mathsf{next}(B))$. If so, then for any immediate predecessor of $\mathsf{next}(A)$, i.e., for any parameter valuation v of B with $u(v) = \mathsf{next}(A)$, monotonicity implies $v < \mathsf{next}(B)$. Formally,

$$\begin{aligned}\mathsf{dep\_met}(B, u, A) \quad = \quad & \mathsf{done}(B) \vee u(\mathsf{next}(B)) \uparrow \\ & \vee\ \mathsf{next}(A) <_T u(\mathsf{next}(B))\end{aligned}$$

Finally, function $\mathsf{update}()$ sets $\mathsf{done}[T]$ to true if the last event for task T has occurred, and otherwise updates the parameters of T according to function $\mathsf{next}(T)$.

Lemma 5.3.4 *For all reachable states s of $\mathcal{N}(\mathcal{A})$ and for all tasks $T \in \mathcal{T}$, the following invariant properties hold:*

1. $v_T^0 \leq_T s.\mathsf{val}(T)$
2. $s.\mathsf{done}[T] \Rightarrow \mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$
3. $s.\mathsf{done}[T] \Rightarrow s.\mathsf{loc}[T] = \mathsf{L1}$

Proof Straightforward by induction on the length of the shortest path leading to s.

Theorem 5.3.5 *Let $\mathcal{A}$ be a PPO. Then LTSs $\mathcal{C}(\mathcal{A})$ and $\mathsf{reach}(\mathcal{N}(\mathcal{A}))$ are isomorphic.*

Proof Let $\mathcal{N}(\mathcal{A}) = (S, s_0, \mathcal{E}, \rightarrow)$. If $s \in S$ is a state and e is an expression containing variables of $\mathcal{N}(\mathcal{A})$, then we write $s.e$ for the result of evaluating expression e in state s. For each event type $A \in \mathcal{E}$, we define a function $\Re_A : S \rightarrow 2^{\mathsf{ev}(\mathcal{A})}$ that associates to each state of $\mathcal{N}(\mathcal{A})$ a set of events of type A. Intuitively, this is the set of events of type A that have occurred before reaching state s. Suppose $\mathsf{task}(A) = T$. Then

$$\begin{array}{rcl} \Re_A(s) & = & \textbf{if } s.\mathsf{done}(A) \textbf{ then} \\ & & \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\} \\ & & \textbf{else} \\ & & \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{next}(A)\} \\ & & \textbf{fi} \end{array}$$

Let function $\Re : S \rightarrow 2^{\mathsf{ev}(\mathcal{A})}$ be defined by:

$$\Re(s) \quad = \quad \bigcup_{A \in \mathcal{E}} \Re_A(s)$$

We will prove that $\Re$ is an isomorphism from $\mathsf{reach}(\mathcal{N}(\mathcal{A}))$ to $\mathcal{C}(\mathcal{A})$.

Claim 1. $\Re(s_0) = \emptyset$.

Proof Let A be an event type. Let $\mathsf{task}(A) = T$. By definition of s_0 we have $s_0.\mathsf{done}(A) = \mathsf{false}$ and $s_0.\mathsf{next}(A) = v_T^0$. Hence, by definition of $\Re_A$, $\Re_A(s_0) = \{(A, v) \mid v <_T v_T^0\}$. But since, by condition **C1**, v_T^0 does not appear in the range of $\mathsf{next}(T)$, there exists no v such that $v <_T v_T^0$. Hence $\Re_A(s_0) = \emptyset$. Since A has been chosen arbitrarily, it follows that also $\Re(s_0) = \emptyset$.

Claim 2. If s is a reachable state and $s \xrightarrow{A} s'$ then $\Re(s) \vdash (A, s.\mathsf{val}(T))$.

Proof Let $v = s.\mathsf{val}(T)$. Assume that $s \xrightarrow{A} s'$ and assume that (B, w) is an immediate predecessor of (A, v). It suffices to prove that $(B, w) \in \Re_B(s)$.

If $\mathsf{task}(B) = \mathsf{task}(A)$ and $A = \mathsf{start}(T)$ then, by **C0**, $B = \mathsf{end}(T)$ and $\mathsf{next}(T)(w) = v$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}[T] = \mathsf{false}$. This implies $s.\mathsf{done}(B) = \mathsf{false}$. Also $s.\mathsf{next}(B) = s.\mathsf{val}(T) = v$. We infer that

$$\Re_B(s) \quad = \quad \{(B, x) \in \mathsf{ev}(\mathcal{A}) \mid x <_T v\}$$

Since $w <_T v$ it follows that $(B, w) \in \Re_B(s)$, as required.

If $\mathsf{task}(B) = \mathsf{task}(A)$ and $A = \mathsf{end}(T)$ then $B = \mathsf{start}(T)$ and $w = v$. If $s.\mathsf{done}(B)$ holds then $(B, w) \in \Re_B(s)$ and we are done. If $s.\mathsf{done}(B)$ does not hold then $\mathsf{next}(T)(\mathsf{val}(T))) \downarrow$ and $\mathsf{next}(B) = \mathsf{next}(T)(\mathsf{val}(T)))$. It follows that $(B, w) \in \Re_B(s)$.

We may, therefore, assume that $\mathsf{task}(B) \neq \mathsf{task}(A)$. Let $U(B,A) = u$ and $\mathsf{task}(B) = T'$. Then, $u(w) = v$. Since $s \xrightarrow{A} s'$, $s.\mathsf{dep\_met}(B,u,A)$ holds. This means that one of the following three cases applies:

- $s.\mathsf{done}(B)$.
  Using the first invariant of Lemma 5.3.4, we infer $v^0_{T'} \leq_{T'} s.\mathsf{val}(T')$. Using the second invariant of Lemma 5.3.4, we infer that $\mathsf{next}(T')(s.\mathsf{val}(T')) \uparrow$. Condition **C3** implies that $v^0_{T'} \leq_{T'} w$. It follows that $w \leq_{T'} s.\mathsf{val}(T')$. Hence $(B,w) \in \Re_B(s)$, as required.

- $s.\mathsf{done}(B) = \mathsf{false}$ and $u(s.\mathsf{next}(B)) \uparrow$.
  By monotonicity imposed by condition **C4**, we do not have $s.\mathsf{next}(B) <_{T'} w$. Condition **C3** implies $v^0_{T'} \leq_{T'} w$, and Lemma 5.3.4 implies $v^0_{T'} \leq_{T'} s.\mathsf{next}(B)$. Hence $w <_{T'} s.\mathsf{next}(B)$ and thus $(B,w) \in \Re_B(s)$.

- $s.\mathsf{next}(A) <_T u(s.\mathsf{next}(B))$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T) = v$. As in the previous case, we use conditions **C3**, **C4** and Lemma 5.3.4 to argue that $w <_{T'} s.\mathsf{next}(B)$, and thus $(B,w) \in \Re_B(s)$.

Claim 3. If $s \xrightarrow{A} s'$ then $\Re(s') = \Re(s) \cup \{(A, s.\mathsf{val}(T))\}$.

Proof Assume $s \xrightarrow{A} s'$. It is easy to check that for all event types B with $\mathsf{task}(B) \neq \mathsf{task}(A)$, $\Re_B(s') = \Re_B(s)$. Let $\overline{\ } : \mathcal{E} \to \mathcal{E}$ be the function given by $\overline{\mathsf{start}(T)} = \mathsf{end}(T)$ and $\overline{\mathsf{end}(T)} = \mathsf{start}(T)$, for all T. We claim that $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$ and $\Re_{\overline{A}}(s') = \Re_{\overline{A}}(s)$. We consider four cases:

- $A = \mathsf{start}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T)$ and $s.\mathsf{done}(A) = \mathsf{false}$. Hence

  $$\Re_A(s) \quad = \quad \{(A,v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$

  Since $s \xrightarrow{A} s'$, $s'.\mathsf{loc}[T] = \mathsf{L2}$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$.
  Since $\mathsf{next}(T)(s'.\mathsf{val}(T)) \uparrow$, then $s'.\mathsf{done}(A)$. Hence

  $$\Re_A(s') \quad = \quad \{(A,v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\}$$

  Thus $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$ and $s'.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$. Moreover $s'.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{val}(T)$. Hence

  $$\begin{array}{rcl} \Re_{\overline{A}}(s') & = & \Re_{\overline{A}}(s) \\ & = & \{(\overline{A},v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\} \end{array}$$

- $A = \mathsf{start}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \downarrow$.
  Since $s \xrightarrow{A} s'$, $s.\mathsf{next}(A) = s.\mathsf{val}(T)$ and $s.\mathsf{done}(A) = \mathsf{false}$. Hence

  $$\Re_A(s) \quad = \quad \{(A,v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$

Since $s \xrightarrow{A} s'$, $s'.\mathsf{loc}[T] = \mathsf{L2}$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Since $\mathsf{next}(T)(s'.\mathsf{val}(T))\downarrow$, then $s'.\mathsf{done}(A) = \mathsf{false}$ and $s'.\mathsf{next}(A) = \mathsf{next}(T)(s'.\mathsf{val}(T))$. Hence

$$\Re_A(s') \quad = \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\}$$

By **C2**, $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. Since $s \xrightarrow{A} s'$, $s.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$ and $s'.\mathsf{done}(\mathsf{end}(T)) = \mathsf{false}$. Moreover $s'.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{next}(\mathsf{end}(T)) = s.\mathsf{val}(T)$. Hence

$$\begin{aligned} \Re_{\overline{A}}(s') &= \Re_{\overline{A}}(s) \\ &= \{(\overline{A}, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\} \end{aligned}$$

- $A = \mathsf{end}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T))\uparrow$.
  Since $s \xrightarrow{A} s'$, $\mathsf{done}(A) = \mathsf{false}$ and $s.\mathsf{next}(A) = s.\mathsf{val}(T)$. Hence

  $$\Re_A(s) \quad = \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$

  Moreover, $s'.\mathsf{done}[T]$, $s'.\mathsf{done}(A)$ and $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$. Hence

  $$\Re_A(s') \quad = \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\}$$

  Thus $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. By the assumptions, $s.\mathsf{done}(\overline{A})$, we can also infer $s'.\mathsf{done}(\overline{A})$. Hence

  $$\begin{aligned} \Re_{\overline{A}}(s') &= \Re_{\overline{A}}(s) \\ &= \{(\overline{A}, v) \in \mathsf{ev}(\mathcal{A}) \mid v \leq_T s.\mathsf{val}(T)\} \end{aligned}$$

- $A = \mathsf{end}(T)$ and $\mathsf{next}(T)(s.\mathsf{val}(T))\downarrow$.
  Since $s \xrightarrow{A} s'$, $\mathsf{done}(A) = \mathsf{false}$ and $s.\mathsf{next}(A) = s.\mathsf{val}(T)$. Hence

  $$\Re_A(s) \quad = \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T s.\mathsf{val}(T)\}$$

  Moreover, $s'.\mathsf{done}(A) = \mathsf{false}$, $s'.\mathsf{next}(A) = s'.\mathsf{val}(T)$ and $s'.\mathsf{val}(T) = \mathsf{next}(T)(s.\mathsf{val}(T))$. Hence

  $$\Re_A(s') \quad = \quad \{(A, v) \in \mathsf{ev}(\mathcal{A}) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\}$$

  By **C2**, $\Re_A(s') = \Re_A(s) \cup \{(A, s.\mathsf{val}(T))\}$. By the assumptions, $s.\mathsf{done}(\overline{A}) = \mathsf{false}$ and $s'.\mathsf{done}(\overline{A}) = \mathsf{false}$. Moreover

  $$s.\mathsf{next}(\overline{A}) = \mathsf{next}(T)(s.\mathsf{val}(T)) = s'.\mathsf{val}(T) = s'.\mathsf{next}(\overline{A})$$

  This implies

  $$\Re_{\overline{A}}(s') \quad = \quad \Re_{\overline{A}}(s)$$

It follows that $\Re(s') = \Re(s) \cup \{(A, s.\mathsf{val}(T))\}$.

Claim 4. If s is a reachable state of $\mathcal{N}(\mathcal{A})$ then $\Re(s) \in \mathsf{conf}(\mathcal{A})$.

Proof Straightforward, by induction on the length of the shortest path to s, using Claims 1-3.

Claim 5. If s, s' are reachable states of $\mathcal{N}(\mathcal{A})$ and $s \xrightarrow{A} s'$ then $\Re(s) \overset{A}{\rightsquigarrow} \Re(s')$.

Proof Straightforward, by combining Claims 2, 3 and 4.

To prove that $\Re$ is bijective, we define an inverse function $\mathfrak{S}$ that maps configurations of $\mathcal{A}$ to states of $\mathcal{N}(\mathcal{A})$. Let C be a configuration and let T be a task. Write C_T for the subset of C of events of type T. We consider four cases:

1. If $C_T = \emptyset$ then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $v_T^0(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

2. If $C_T \neq \emptyset$ and the unique maximal event of C_T (cf Lemma 5.3.3) is of the form $(\mathsf{start}(T), v)$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L2}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $v(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

3. If $C_T \neq \emptyset$, the unique maximal event of C_T is of the form $(\mathsf{end}(T), v)$ and $\mathsf{next}(T)(v)\downarrow$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to false, and variable $T.\mathsf{p}$ is set to $\mathsf{next}(T)(v)(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

4. If $C_T \neq \emptyset$, the unique maximal event of C_T is of the form $(\mathsf{end}(T), v)$ and $\mathsf{next}(T)(v)\uparrow$, then variable $\mathsf{loc}[T]$ is set to $\mathsf{L1}$, variable $\mathsf{done}[T]$ is set to true, and variable $T.\mathsf{p}$ is set to $v(\mathsf{p})$, for each parameter $\mathsf{p} \in \mathcal{M}(T)$.

The following claim directly implies that $\Re$ is injective.

Claim 6. For each reachable state s of network $\mathcal{N}(\mathcal{A})$, $\mathfrak{S}(\Re(s)) = s$.

Proof Assume s is a reachable state of $\mathcal{N}(\mathcal{A})$. Let $C = \Re(s)$ and $s' = \mathfrak{S}(C)$. We must prove $s' = s$. Assume $T \in \mathcal{T}$. It suffices to prove, $s'.\mathsf{val}(T) = s.\mathsf{val}(T)$, $s'.\mathsf{loc}[T] = s'.\mathsf{loc}[T]$ and $s'.\mathsf{done}[T] = s'.\mathsf{done}[T]$. Let $A = \mathsf{start}(T)$ and $B = \mathsf{end}(T)$. We consider five cases:

1. $s.\mathsf{done}[T] = \mathsf{false}$ and $s.\mathsf{loc}[T] = \mathsf{L1}$ and $s.\mathsf{val}(T) = v_T^0$. Then, by Claim 1, $C = \emptyset$. Hence, also $C_T = \emptyset$. By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = v_T^0$. Thus, $s' = s$, as required.

2. $s.\mathsf{done}[T] = \mathsf{false}$ and $s.\mathsf{loc}[T] = \mathsf{L1}$ and $s.\mathsf{val}(T) \neq v_T^0$. Then $s.\mathsf{done}(A) = s.\mathsf{done}(B) = \mathsf{false}$, so

$$C_T \quad = \quad \{(A, v), (B, v) \mid v <_T s.\mathsf{val}(T)\}.$$

   By Lemma 5.3.4, $v_T^0 \leq_T s.\mathsf{val}(T)$. Hence, by assumption $s.\mathsf{val}(T) \neq v_T^0$, $v_T^0 <_T s.Val(T)$. Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is of the form (B, w) with $\mathsf{next}(T)(w) = s.\mathsf{val}(T)$). By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.

3. $s.\mathsf{done}[T] = \mathsf{false}$, $s.\mathsf{loc}[T] = \mathsf{L2}$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$. Then $s.\mathsf{done}(A) = \mathsf{true}$ and $s.\mathsf{done}(B) = \mathsf{false}$, so

$$C_T \quad = \quad \{(A, v) \mid v \leq_T s.\mathsf{val}(T)\} \cup \{(B, v) \mid v <_T s.\mathsf{val}(T)\}$$

Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is $(A, s.\mathsf{val}(T))$. Hence, by definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L2}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.

4. $s.\mathsf{done}[T] = \mathsf{false}$, $s.\mathsf{loc}[T] = \mathsf{L2}$ and $\mathsf{next}(T)(s.\mathsf{val}(T)) \downarrow$. Then $s.\mathsf{done}(A) = s.\mathsf{done}(B) = \mathsf{false}$ and

$$\begin{aligned} C_T \quad = \quad & \{(A, v) \mid v <_T \mathsf{next}(T)(s.\mathsf{val}(T))\} \\ & \cup \{(B, v) \mid v <_T s.\mathsf{val}(T)\} \end{aligned}$$

Thus $C_T \neq \emptyset$ and the unique maximal event of C_T is $(A, s.\mathsf{val}(T))$. Hence, by definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L2}$, $s'.\mathsf{done}[T] = \mathsf{false}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus, $s' = s$, as required.

5. $s.\mathsf{done}[T] = \mathsf{true}$. Then, by definition of $\Re$, $C_T = \{(A, v), (B, v) \mid v \leq_T s.\mathsf{val}(T)\}$. Hence $C_T \neq \emptyset$ and the unique maximal event of C_T is $(B, s.\mathsf{val}(T))$. By Lemma 5.3.4, $\mathsf{next}(T)(s.\mathsf{val}(T)) \uparrow$ and $s.\mathsf{loc}[T] = \mathsf{L1}$. By definition of $\mathfrak{S}$, $s'.\mathsf{loc}[T] = \mathsf{L1}$, $s'\mathsf{done}[T] = \mathsf{true}$ and $s'.\mathsf{val}[T] = s.\mathsf{val}[T]$. Thus $s' = s$, as required.

Claim 7. If s is reachable, $\Re(s) = C$, $C \overset{A}{\rightsquigarrow} C'$ and $s' = \mathfrak{S}(C')$ then $s \xrightarrow{A} s'$.

Proof By Claim 6, $\mathfrak{S}(C) = s$. Let $\mathsf{task}(A) = T$. Since $C \overset{A}{\rightsquigarrow} C'$, $s.\mathsf{done}[T] = \mathsf{false}$. Hence, to prove that s enables an A-transition, it suffices to establish that $\mathsf{dep\_met}(A)$ holds in s. For this, in turn, it suffices to prove, for any incoming precedence edge $B \overset{u}{\rightarrow} A$ with $\mathsf{task}(B) \neq \mathsf{task}(A)$, that $\mathsf{dep\_met}(B, u, A)$ holds in s. Let $C' = C \cup \{\alpha\}$ with $\alpha = (A, v)$. Since $C \vdash \alpha$, all immediate predecessors of α are in C. Let $B \overset{u}{\rightarrow} A$ be a precedence edge of A and let $\mathsf{task}(B) = T'$. We consider the following cases:

- $C_{T'} = \emptyset$
  Since C contains all immediate predecessors of α, there exists no event (B, w) such that $U(B, A)(w) = v$. Therefore, since $C_{T'} = \emptyset$, then $s.\mathsf{done}[T'] = \mathsf{false}$ and $s.\mathsf{loc}[T'] = \mathsf{L1}$, it means that $s.\mathsf{done}(B) = \mathsf{false}$. Knowing that $B \overset{u}{\rightarrow} A$ and $s.\mathsf{done}(B) = \mathsf{false}$, the next event of type B, namely $\beta = (B, v^0_{T'})$ will occur in future. If $u(\mathsf{next}(B)) \downarrow$ and β is not an immediate predecessor of α, it follows that $v < u(\mathsf{next}(B))$, meaning that $\mathsf{dep\_met}(B, u, A)$ holds in s. If $u(\mathsf{next}(B)) \uparrow$, by the second case in the definition of $\mathsf{dep\_met}(B, u, A)$, $\mathsf{dep\_met}(B, u, A)$ is true in s.

- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ of the form $(\mathsf{end}(T), w)$ with $\mathsf{next}(T')(w) \uparrow$. This implies that $s.\mathsf{done}[T'] = \mathsf{true}$ and that $s.\mathsf{done}(B)$ holds, therefore, $\mathsf{dep\_met}(B, u, A)$ holds is s.

- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ where $\mathsf{next}(T')(w)\downarrow$ and $u(s.\mathsf{next}(B))\uparrow$. This means that the second condition in the definition of $\mathsf{dep\_met}(B, u, A)$ is true, meaning that $\mathsf{dep\_met}(B, u, A)$ holds is s.

- $C_{T'} \neq \emptyset$ and (B, w) is the unique maximal event of $C_{T'}$ where $\mathsf{next}(T')(w)\downarrow$ and $u(s.\mathsf{next}(B))\downarrow$. Since $u(B, A)(w) = v$ it means that $u(s.\mathsf{next}(B)) \neq_T v$, and by **C4**, we have that $u(w) <_T u(s.\mathsf{next}(B))$, therefore, $\mathsf{dep\_met}(B, u, A)$ holds in s.

We conclude that s enables an A-transition. Suppose $s \xrightarrow{A} s''$. Then, by Claim 5, $\Re(s) \overset{A}{\rightsquigarrow} \Re(s'')$. Since C has only one outgoing A-transition, $\Re(s'') = C'$. Hence, by Claim 6, $s'' = s'$, as required.

Claim 8. $\Re$ is a bijection from the reachable states of $\mathcal{N}(\mathcal{A})$ to $\mathsf{conf}(\mathcal{A})$.

Proof Straightforward using Claims 1, 4, 6 and 7.

The theorem now follows by combination of the claims.

5.4 Real-Time Task Systems

In the previous section, we have introduced PPOs as a compact representation of task graphs with repetitive behavior. PPOs provide a convenient formalism to model embedded applications, but they need to be incorporated in a larger formalism that supports modeling of the execution platform and of the mapping from applications onto this platform. In this section, we introduce such a formalism, which we name *real-time task systems (RTTS)*, and describe how the translation of PPOs to Uppaal from Section 5.3 can be lifted to RTTSs. Our notion of a RTTS has been very much inspired by the modeling language that is supported by the Octopus DSEIR toolset [Oct11, BVBG+10].

5.4.1 Definition of RTTS

In an RTTS, an application is modeled using a PPO $\mathcal{A}$ together with a function w that gives the size of each task. The platform is modeled abstractly using a set $\mathcal{R}$ of resources and a function cap. Resources in $\mathcal{R}$ can be anything ranging from CPUs, memory, communication bandwidth and dedicated processing blocks, to devices such as scanners and printers. Each resource r has $\mathsf{cap}(r)$ units available, e.g., 3 CPUs, $133MB$ memory, and $10Mb/s$ bandwidth. The mapping from application to platform is specified using functions cl, h and ρ. Function cl specifies, for each task T and resource r, a bound of the number of units of r that can be allocated to T. In practice, resources are often handed over from one task to another, for instance when one task has created a file that is being processed further by another task. This handover of resources is specified by function h. The size of each task

is specified via a function w. Finally, function ρ specifies the pace at which a task progresses, given the resources that have been allocated to it.

A *real-time task system (RTTS)* is a tuple $\mathcal{RTTS} = (\mathcal{A}, \mathcal{R}, \mathsf{cap}, \mathsf{cl}, \mathsf{h}, \mathsf{w}, \rho)$, where [2]

- $\mathcal{A} = (\mathcal{T}, \mathcal{M}, E, U)$ is a PPO.
- $\mathcal{R}$ is a finite set of resources.
- $\mathsf{cap} : \mathcal{R} \to \mathbb{N}$ is a function that specifies for each resource the total number of units that are available.
- $\mathsf{cl} : \mathcal{T} \to (\mathcal{R} \to \mathbb{N})$ is a function that specifies, for each resource, the maximum number of units that a task may claim. A task cannot claim more resources than what is available: for each $T \in \mathcal{T}$, $\mathsf{cl}(T) \leq \mathsf{cap}$.
- $\mathsf{h} : E \to (\mathcal{R} \to \mathbb{N})$ specifies the resources handed over from one task to another via edges of the PPO. We require that resources may be handed over only to start events: for all $A \in \mathcal{E}$ and $T \in \mathcal{T}$,

$$\mathsf{h}(A, \mathsf{end}(T)) \quad = \quad \mathbf{0},$$

  where $\mathbf{0} : \mathcal{R} \to \mathbb{N}$ is given by $\mathbf{0}(r) = 0$, for $r \in \mathcal{R}$.
- $\mathsf{w} : \mathcal{T} \to \mathbb{N}$ is a function that specifies the *size* of each task, i.e., the amount of work that has to be done.
- $\rho : \mathcal{T} \times (\mathcal{R} \to \mathbb{N}) \to \mathbb{N}$ is a function that specifies the *pace* at which each task is processed, given the resources that have been assigned to it. We require that the pace increases monotonically with the number of resources available: for all $T \in \mathcal{T}$ and $a, a' : \mathcal{R} \to \mathbb{N}$,

$$a \leq a' \quad \Rightarrow \quad \rho(T, a) \leq \rho(T, a').$$

  However, the pace will not increase any further once the maximum number of resources that a task may claim has been allocated: $\rho(T, a) \leq \rho(T, \mathsf{cl}(T))$.

We call a resource r *static* if each task in the system may only progress when its maximum claim for resource r has been assigned. Formally, $r \in \mathcal{R}$ is *static* if, for all $T \in \mathcal{T}$ and for all $a : \mathcal{R} \to \mathbb{N}$,

$$a \leq \mathsf{cl}(T) \wedge \rho(T, a) > 0 \quad \Rightarrow \quad a(r) = \mathsf{cl}(T)(r).$$

Resources that are not static are called *dynamic*. A task that uses a dynamic resource may run faster if we assign more units of this resource to it. A typical example of a dynamic resource is communication bandwidth.

[2] In this section, we use functions of type $\mathcal{R} \to \mathbb{N}$, where $\mathcal{R}$ is some set of resources. Operations and predicates on $\mathbb{N}$ are extended to such function by pointwise extension. For instance, for $f, g : \mathcal{R} \to \mathbb{N}$, we say that $f \leq g$ iff $\forall r \in \mathcal{R} : f(r) \leq g(r)$, and we define $f + g : \mathcal{R} \to \mathbb{N}$ by $(f + g)(r) = f(r) + g(r)$.

5.4.2 Semantics of RTTS

A *task instance* of PPO $\mathcal{A}$ is a pair $\beta = (T, v)$, where $T \in \mathcal{T}$ and $v \in V(T)$. We write $\mathsf{task}(\beta) = T$ and use $\mathsf{start}(\beta)$ and $\mathsf{end}(\beta)$ to denote the start and end event, respectively, that correspond to β. We let $\mathsf{ti}(\mathcal{A})$ denote the set of task instances of PPO $\mathcal{A}$. We say that a task instance β is *done* in configuration C if $\mathsf{end}(\beta) \in C$, we say that β is *active* in C if $\mathsf{start}(\beta) \in C$ and $\mathsf{end}(\beta) \notin C$, and we say that β is *waiting* in C if $\mathsf{start}(\beta) \notin C$. Clearly, each task instance is either done, active or waiting, in any given configuration C. We define:

$$\begin{aligned} \mathsf{done}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{end}(\beta) \in C\}, \\ \mathsf{active}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{start}(\beta) \in C \wedge \mathsf{end}(\beta) \notin C\}, \\ \mathsf{waiting}(C) &= \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \mathsf{start}(\beta) \notin C\}. \end{aligned}$$

A timed configuration records the precise global state of the modeled system at some point during execution, i.e., the set of events that have occurred, the resources that have been allocated to each task instance, and the completion level of each task instance. Formally, a *timed configuration* of $\mathcal{RTTS}$ is a triple (C, O, θ) where

- $C \in \mathsf{conf}(\mathcal{A})$.
- $O : \mathsf{ti}(\mathcal{A}) \to (\mathcal{R} \to \mathbb{N})$ specifies allocation of resources to task instances. We require that O does not allocate more resources than what is available in total: $\sum_{\gamma \in \mathsf{ti}(\mathcal{A})} O(\gamma) \leq \mathsf{cap}$. Moreover, we require, for each $\beta \in \mathsf{ti}(\mathcal{A})$ with $\mathsf{task}(\beta) = T$,

  1. O does not allocate more resources to β than what T may claim: $O(\beta) \leq \mathsf{cl}(T)$.
  2. If β is waiting in C, the only resources allocated to β are those that have been handed over by preceding events:

  $$\beta \in \mathsf{waiting}(C) \Rightarrow O(\beta) = \sum_{\alpha \in C \mid \alpha \mapsto \mathsf{start}(\beta)} \mathsf{h}(\mathsf{ev\_type}(\alpha), \mathsf{start}(T))$$

  3. If β is active in C, enough resources are allocated to β for the handover to its successors upon termination:

  $$\beta \in \mathsf{active}(C) \Rightarrow O(\beta) \geq \sum_{\alpha \in \mathsf{ev}(\mathcal{A}) \mid \mathsf{end}(\beta) \mapsto \alpha} \mathsf{h}(\mathsf{end}(T), \mathsf{ev\_type}(\alpha))$$

  4. If β is done in C, no resources are allocated to it: $\beta \in \mathsf{done}(C) \Rightarrow$

  $$O(\beta) = \mathbf{0}.$$

$$\frac{C \xrightarrow{\mathsf{start}(\beta)} C'}{(C, O, \theta) \xrightarrow{\mathsf{start}(\beta)} (C', O', \theta)}$$

$$\frac{\begin{array}{c} d \in \mathbb{R}_{\geq 0} \\ \forall \gamma \notin \mathsf{active}(C) : \theta'(\gamma) = \theta(\gamma) \\ \forall \gamma \in \mathsf{active}(C) : \theta'(\gamma) = \theta(\gamma) - d \cdot \rho(\mathsf{task}(\gamma), O(\gamma)) \end{array}}{(C, O, \theta) \xrightarrow{d} (C, O, \theta')}$$

$$\frac{C \xrightarrow{\mathsf{end}(\beta)} C'}{(C, O, \theta) \xrightarrow{\mathsf{end}(\beta)} (C', O', \theta)}$$

Figure 5.3: Semantics of real-time task systems

- $\theta : \mathsf{ti}(\mathcal{A}) \to \mathbb{R}_{\geq 0}$ specifies for each task instance the amount of work that remains to be done. We require, for each $\beta \in \mathsf{ti}(\mathcal{A})$,

$$\begin{array}{rcl} 0 \leq \theta(\beta) & \leq & \mathsf{w}(\mathsf{task}(\beta)) \\ \beta \in \mathsf{done}(C) & \Rightarrow & \theta(\beta) = 0 \\ \beta \in \mathsf{waiting}(C) & \Rightarrow & \theta(\beta) = \mathsf{w}(\mathsf{task}(\beta)) \end{array}$$

The *timed configuration structure* of $\mathcal{RTTS}$ is the LTS $\mathcal{TC}(\mathcal{RTTS})$ with as states the timed configurations of $\mathcal{RTTS}$, as initial state the timed configuration $(\emptyset, O_0, \theta_0)$, where, for all β, $O_0(\beta) = \mathbf{0}$ and $\theta_0(\beta) = \mathsf{w}(\mathsf{task}(\beta))$, as actions the set $\mathsf{ev}(\mathcal{A}) \cup \mathbb{R}_{\geq 0}$, and a transition relation that is defined by the rules in Figure 5.3. These rules describe how $\mathcal{RTTS}$ may evolve from timed configuration (C, O, θ) to timed configuration (C', O', θ') due to the occurrence of an event or through passage of time.

Note that the rules of Figure 5.3 do not impose any relationship between the resource allocations before and after an event. In fact, a priori we allow for a complete reshuffling of the resource allocation whenever an event occurs. Also, there are no constraints on the time at which a new task instance starts. In practice, of course, we need to impose restrictions on timing and on how resource allocations may change. This is achieved using so-called *scheduling rules*, which we will discuss in the next subsection.

5.4.3 Scheduling Rules

The timed configuration structures defined in the previous subsection are nondeterministic LTSs; after each event, there may be an arbitrary, complete reallocation of resources between tasks. Also, the choice when to start a new task instance

is entirely left open (there can be an arbitrary delay). We allow for delays and reallocation of resources in our semantics because this is what happens in the embedded systems that we model. However, in applications we will typically adopt a number of *scheduling rules* that severely reduce (or even eliminate) the nondeterminism within a real-time task system, and impose constraints on the timing of events. Effectively, these rules cut away certain transitions and configurations in the semantics of the real-time task system that we have defined in Figure 5.3. Below we give two examples of commonly used, generic scheduling rules. Additional scheduling rules may be defined for each application.

1. *Nonlaziness*
   A task may be *lazy* or *nonlazy*. Semantically, this means that we remove all the time passage transitions $(C, O, \theta) \xrightarrow{d} (C', O', \theta')$, such that $d > 0$ and an event $\mathsf{start}(\beta)$ with $\mathsf{task}(\beta)$ nonlazy is enabled in (C, O, θ). In this way, we express that no time delay is allowed whenever an instance of a nonlazy task is enabled and sufficient resources are available: immediately either this or a competing task will start.

2. *Preemption*
   A task may be *preemptive* or *non-preemptive*. An instance of a preemptive task may be interrupted while running (i.e., the pace may become 0 due to resources that are taken away), while this is not possible for any instance of a non-preemptive task. Semantically, we reduce the timed configuration structure by removing all configurations (C, O, θ) with $\rho(\mathsf{task}(\gamma), O(\gamma)) = 0$, for some event instance $\gamma \in \mathsf{active}(C)$ with $\mathsf{task}(\gamma)$ non-preemptive.

Example Figures 5.4-5.6 depict, in a graphical manner, an RTTS that models a printer application. This case study will also be used in Section 5.6 for the framework performance evaluation. The underlying PPOs are depicted using the notational conventions explained already in Example 5.3.1. For readability, we have omitted the standard edge from e to s, for each of the tasks. We assume that all tasks are nonlazy and non-preemptive. The ovals encode resources and the parentheses contain their maximum capacity available (one if not mentioned). All the resources are static, except for USB up and USB down. The dynamic resources USB up and USB down obey the additional scheduling rule:

> **USB**: A timed configuration (C, O, θ) is only allowed if, when $O(\alpha)(\mathsf{USB\ up})$ and $O(\beta)(\mathsf{USB\ down})$ are both positive, for some task instances α and β, then they are both equal to 3 Mb/s. If only one is positive then it is equal to 4 Mb/s.

The idea is that if two processes use the bus simultaneously, transmission takes place at a lower pace.

The dashed lines between resources and tasks indicate resource claims. Precedence edges are annotated with the number of resource units handed over. The

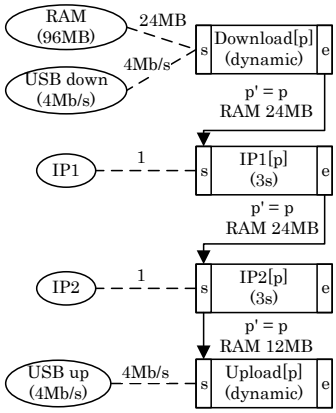


Figure 5.4: Process from Store RTTS.

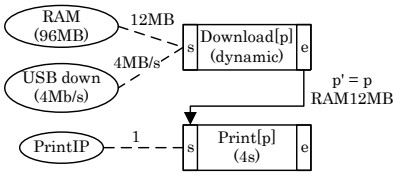


Figure 5.5: Simple Print RTTS.

total resource claim of a task can be obtained by taking the resource claims indicated by dotted lines plus all the resources that are handed over via incoming precedence edges of the start event minus all the resources that are handed over via outgoing precedence edges of the start event. Thus, for instance, task IP1 in Figure 5.4 claims 1 unit of resource IP1 and 24 units of resource RAM. Task Scan in Figure 5.6 claims 1 unit of resource Scanner and 0 units of resource RAM.

The rectangles denoting tasks may contain, between parentheses, task durations. If a task T is annotated with a task duration of t seconds then this means that the total amount of work is $n \times t$ units and the pace is n units per second if the task has all the resources that it claims, and 0 units per second otherwise. For tasks that use dynamic resource USB up, the duration is determined by the expression dynamic that is defined by:

$$\begin{aligned} \mathsf{dynamic} \quad \equiv \quad & \text{if USB up } = 4Mb/s \text{ then } 3s \text{ else} \\ & \text{if USB up } = 3Mb/s \text{ then } 4s \text{ else } 0s. \end{aligned}$$

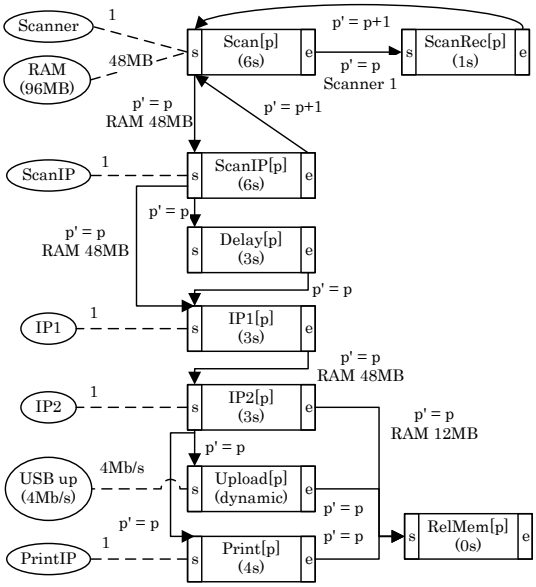


Figure 5.6: Direct Copy RTTS.

For tasks that use dynamic resource USB down, the corresponding predicate is defined by:

$$\begin{aligned} \mathsf{dynamic} \quad \equiv \quad & \text{if USB down } = 4Mb/s \text{ then } 3s \text{ else} \\ & \text{if USB down} = 3Mb/s \text{ then } 4s \text{ else } 0s. \end{aligned}$$

For these tasks, the total amount of work is $12Mb$ and the pace is either $4Mb/s$ or $3Mb/s$. The reader may verify that the timed configuration structure of this RTTS is a deterministic LTS. In particular, after each event the allocation of resources to task instances is uniquely determined.

5.5 Generated UPPAAL Models

We have implemented a tool that generates UPPAAL models for a subclass of RTTSs. Our tool supports the specification of lazy and nonlazy tasks, and of preemptive and non-preemptive tasks. It accepts any $\mathcal{RTTS} = (\mathcal{A}, \mathcal{R}, \mathsf{cap}, \mathsf{cl}, \mathsf{h}, \mathsf{w}, \rho)$ in which:

1. $\mathcal{A}$ is a restricted PPO such that, for each $T \in \mathcal{T}$, the only incoming edge of $\mathsf{end}(T)$ comes from $\mathsf{start}(T)$.
   This assumption allows us to obtain a translation of RTTSs to UPPAAL by extending the translation of PPOs to UPPAAL that we defined in Section 5.3.

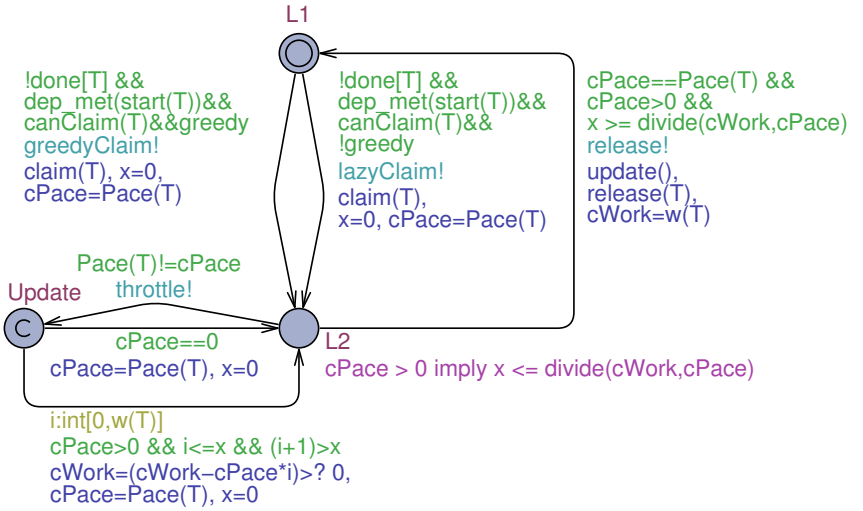


Figure 5.7: Template for RTTS-based task automata.

2. Handover of resources from task T to task T' is only allowed if the precedence function for the corresponding PPO edge is bijective.
   This assumption simplifies the treatment of handovers, since whenever an instance of a task T is enabled, i.e., all its immediate predecessors are done, the total number of resources that has been handed over to this task instance is always:
   $$\sum_{A \in \mathcal{E} \mid (A, \mathsf{start}(T)) \in E} \mathsf{h}(A, \mathsf{start}(T)).$$

3. Additional scheduling rules enforce that, for each timed configuration, the resource allocation is uniquely determined by the configuration of the PPO. Formally, we require that there exists a function $\mathsf{alloc} : \mathsf{conf}(\mathcal{A}) \to (\mathsf{ti}(\mathcal{A}) \to (\mathcal{R} \to \mathbb{N}))$ such that, for each timed configuration (C, O, θ), (C, O, θ) is allowed by the scheduling rules if and only if $O = \mathsf{alloc}(C)$. We require that $\mathsf{alloc}(\emptyset) = O_0$ and that, for each configuration C, $O = \mathsf{alloc}(C)$ satisfies the five conditions on O required for timed configurations.
   This assumption simplifies the translation since it eliminates the need to record, in each state, the exact allocation of resources to all the task instances.

It is easy to verify that the RTTS described in the example of Section 5.4.3 satisfies the above conditions. In particular, the resource allocation O is uniquely determined by the configuration C of the PPO: (1) For a task instance β that is waiting in C, the only resources allocated are those that have been handed over by preceding task instances. (2) By definition of a timed configuration, no resources can be allocated to a task instance β that is done in C. (3) If β is active in C, then

for each static resource r, $O(\beta)(r) = \mathsf{cl}(\mathsf{task}(\beta))(r)$ (all tasks in the example are non-preemptive), and for USBup and USBdown the resource allocation is uniquely determined by rule **USB**.

Let $\mathcal{RTTS}$ be an RTTS and alloc be a resource allocation function that satisfy the above restrictions. We define $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ as the LTS induced by the parallel composition of the UPPAAL timed automaton shown in Figure 5.7, for each $T \in \mathcal{T}$. Below we explain the various predicates and functions used in this timed automaton. The automaton uses the same global shared variables as the automaton for PPOs described in Section 5.3:

$$\{T.\mathsf{p}, \mathsf{loc}[T], \mathsf{done}[T] \mid T \in \mathcal{T} \wedge \mathsf{p} \in \mathcal{M}(T)\}.$$

As in Section 5.3, variable $T.\mathsf{p}$ gives the value of p in the next event of T that will occur. Variable $\mathsf{loc}[T]$ records whether the automaton is in location L1 or L2. Even though there is now an extra location Update in the automaton, the domain of $\mathsf{loc}[T]$ remains unchanged: during the brief excursions to location Update, variable $\mathsf{loc}[T]$ keeps value L2. As in Section 5.3, Boolean variable $\mathsf{done}[T]$ records whether the last event of T has been executed.

In addition, the timed automaton maintains three local variables: integer cWork records the latest estimate of the work that remains to be done, integer cPace records the current pace of task T, and clock x records the time that has elapsed since the start of the task or the last change of pace. Initially $\mathsf{cWork} = \mathsf{w}(T)$, $\mathsf{cPace} = 0$ and $\mathsf{x} = 0$. Finally, for each resource r, variable $\mathsf{resource\_cap}[r]$ records the number of units of resource r that are still available. Initially, $\mathsf{resource\_cap}[r] = \mathsf{cap}(r)$, for each r.

The automaton shown in Figure 5.7 has the same structure as the automaton of Figure 5.2 in Section 5.3 and uses exactly the same functions dep_met and update. However, to handle resource allocation and timing, the automaton is equipped with some additional structure. Predicate $\mathsf{canClaim}(T)$ checks whether enough resources are available to start the next instance of task T:

$$\mathsf{canClaim}(T) \;=\; \rho(T, \mathsf{resource\_cap} + \sum_{A \in \mathcal{E} \mid (A, \mathsf{start}(T)) \in E} \mathsf{h}(A, \mathsf{start}(T)) - \sum_{B \in \mathcal{E} \mid (\mathsf{start}(T), B) \in E} \mathsf{h}(\mathsf{start}(T), B)) > 0$$

Predicate $\mathsf{canClaim}(T)$ evaluates to true and the next instance of T may start if enough resources can be allocated such that, together with the resources that have been handed over make the pace positive.

Parameter greedy specifies if task T is nonlazy (true) or lazy (false). The automaton has two transitions from location L1 to L2. If task T is nonlazy we take the transition labeled with urgent broadcast channel greedyClaim, whereas if task T is lazy we take the transition labeled with (nonurgent) broadcast channel lazyClaim. In this way, we ensure that a nonlazy task starts as soon as it becomes enabled, whereas the start of a lazy task may be postponed.

Function $\mathsf{claim}(T)$ first sets the location variable $\mathsf{loc}[T]$ to $\mathsf{L2}$. Let s be the resulting global state of the UPPAAL model. We may then compute the corresponding PPO-configuration by applying[3] the function $\Re$ introduced in the proof of Theorem 5.3.5: $C = \Re(s)$. Next, we may compute the resource allocation function associated to C: $O = \mathsf{alloc}(C)$. Once we know which resources have been assigned, function claim computes which resources are still available in the new state and updates the value of $\mathsf{resource\_cap}$:

$$\mathsf{resource\_cap} \quad = \quad \mathsf{cap} - (\sum_{\gamma \in \mathsf{ti}(\mathcal{A})} O(\gamma))$$

Function $\mathsf{Pace}(T)$ computes the pace of the task instance of T that is currently active. If s is the current global state of the model then

$$\mathsf{Pace}(T) \quad = \quad \rho(T, \mathsf{alloc}(\Re(s))(T, s.\mathsf{val}(T)))$$

Whenever the automaton starts a new instance of task T by jumping from location $\mathsf{L1}$ to location $\mathsf{L2}$, it resets clock x and sets variable cPace to $\mathsf{Pace}(T)$. Assuming the pace remains unchanged, the time needed to complete the task instance is $\frac{\mathsf{cWork}}{\mathsf{cPace}}$. Since in UPPAAL we may only compare clocks to integers, we sometimes need to slightly overapproximate the task duration: the automaton may leave location $\mathsf{L2}$ when $\mathsf{x} == \mathsf{divide}(\mathsf{cWork}, \mathsf{cPace})$. Here function divide divides its two arguments and takes the ceiling.

At any point during execution of an instance of task T, due to the start or completion of some other task instance, the allocation of resources and hence the pace of T may change. Whenever this happens, the automaton for task T instantly jumps to location Update via a transition labelled with urgent broadcast channel $\mathsf{throttle}$. From location Update the automaton instantly jumps back to location $\mathsf{L2}$.[4] Since Update is committed, no other automaton may perform a transition in between. There are two cases. If task T was preempted ($\mathsf{cPace} == 0$), the amount of work to be done remains unchanged: the automaton sets cPace to the new value, resets clock x, and jumps to $\mathsf{L2}$. If task T was running ($\mathsf{cPace} > 0$) then the remaining amount of work is $\mathsf{cWork} - \mathsf{cPace} * \mathsf{x}$. Since UPPAAL does not permit the use of clock variables in integer expressions, we (conservatively) approximate this value. Using a select statement, we pick the largest integer i that does not exceed x and decrement cWork with $\mathsf{cPace} * \mathsf{i}$. In addition, we set cPace to the new value, reset clock x, and jump to $\mathsf{L2}$.

A task instance may end and the automaton may jump back from $\mathsf{L2}$ to $\mathsf{L1}$ when the work has been completed: since we assume that in the PPO the only incoming edge of $\mathsf{end}(T)$ comes from $\mathsf{start}(T)$, we do not need to check anymore

[3] Formally, function $\Re$ takes as input global states of the UPPAAL model described in Section 5.3. We may apply $\Re$ to global states of the extended model described in this section by removing all the additional state variables.

[4] The intermediate location Update is required since in UPPAAL clock guards are not allowed on edges labeled with urgent channels.

if dep_met(end(T)) holds. On the transition from L2 to L1, function update() (introduced in Section 5.3) sets done[T] to true if the last event for task T has occurred and otherwise updates the parameters of T. Variable cWork is reinitialized to w(T), and function release(T) sets loc[T] to L1, computes which resources are still available in the new state and updates the value of resource_cap accordingly, in exactly the same way as function claim(T).

Of course, we would like to prove that, for any $\mathcal{RTTS}$ and resource allocation alloc that satisfies our constraints, the LTS $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ induced by our UPPAAL translation is behaviorally equivalent with the timed configuration structure $\mathcal{TC}(\mathcal{RTTS})$, pruned according to function alloc. Unfortunately, due to the throttle transitions in the UPPAAL model and the overapproximations which they induce, there is no exact correspondence. Previous experiments with this type of approximations [IKY+08, IV10] suggest that in practical applications (from the printing domain) the errors that they introduce are minimal, but it remains to be investigated if (and how) these observations can be formalized using some notion of schedule robustness.

Below we establish that in the special case in which all tasks are non-preemptive and all resources are static, and moreover work divided by pace is always an integer, there is an exact correspondence between the semantics of an RTTS and the semantics of its UPPAAL translation. It is not possible to prove that the two structures are isomorphic: from a timed configuration we cannot infer a unique value for the local clock of a task automaton in location L1. Instead, we establish a bisimulation between the two semantic structures.

Theorem 5.5.1 *Let $\mathcal{RTTS}$ be a real-time task system, with all resources static and all tasks non-preemptive, let* alloc *be a function that maps configurations to resource allocations, and assume $\mathcal{RTTS}$ and* alloc *satisfy the constraints listed at the beginning of this section. Assume further that, for each task T and resource allocation a with $\rho(T,a) > 0$, $\mathsf{w}(T)/\rho(T,a)$ is an integer. Then the LTS $\mathcal{TC}(\mathcal{RTTS})$, pruned according to* alloc*, and the LTS $\mathcal{N}(\mathcal{RTTS}, \mathsf{alloc})$ are bisimilar.*

Proof Since all tasks of $\mathcal{RTTS}$ are non-preemtive, an active task instance always has a positive pace. Moreover, since all tasks of $\mathcal{RTTS}$ are static,

$$a \leq \mathsf{cl}(T) \wedge \rho(T,a) > 0 \quad \Rightarrow \quad a = \mathsf{cl}(T).$$

This means that the pace of an active instance of task T is constant and always equal to $\rho(T, \mathsf{cl}(T))$. Hence, in a run of the UPPAAL model location, the automaton for any task T will never reach location Update. Hence, if s is a reachable state of the UPPAAL semantics with $s.\mathsf{loc}[T] = \mathsf{L2}$, then $s.T.\mathsf{cPace} = \rho(T, \mathsf{cl}(T))$ and $s.T.\mathsf{cWork} = \mathsf{w}(T)$.

Let s be a reachable state of the UPPAAL semantics. We define $f(s)$ to be the timed configuration (C, O, θ) where $C = \Re(s)$, $O = \mathsf{alloc}(C)$ and, for each

$\beta \in \mathsf{ti}(\mathcal{A})$ with $\mathsf{task}(\beta) = T$,

$$\theta(\beta) \quad = \quad \begin{cases} \mathsf{w}(T) & \text{if } \beta \in \mathsf{waiting}(C) \\ \mathsf{w}(T) - s.T.\mathsf{x} \cdot \rho(T, \mathsf{cl}(T)) & \text{if } \beta \in \mathsf{active}(C) \\ 0 & \text{if } \beta \in \mathsf{done}(C) \end{cases}$$

Observe that (C, O, θ) is a timed configuration since by Claim 4 in the proof of Theorem 5.3.5, $C = \Re(s)$ is a configuration; by the assumption about alloc, $O = \mathsf{alloc}(C)$ satisfies the five conditions for O required for timed configuration, and by construction θ also satisfies the conditions required for a timed configuration. Since $O = \mathsf{alloc}(C)$, the timed configuration is allowed by the scheduling rules.

It is routine to check that the relation R that relates s and $f(s)$, for each reachable state of the UPPAAL semantics, is a bisimulation relation.

5.6 Experiments

We now turn to an experimental evaluation of UPPAAL models generated from the RTTS representation. We compare these models with handcrafted models that have been presented in Chapter 2. In the handcrafted models, each RTTS is modeled as a single automaton that contains all tasks. This way of modeling is more natural for design engineers but less efficient to analyze as proven below. The case study used for the comparison of the two modeling methods is taken from Section 2.2. The printer architecture studied here is depicted in Figure 2.1 with the RTTSs depicted in Figures 5.4-5.6. We computed for each experiment the fastest time in which all tasks were completed (also called makespan) assuming that all tasks were nonlazy. All experiments were performed with UPPAAL, version 4.1.2, on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3 GHz) and 128 Gb of DDR2 RAM.

Three performance metrics were used to evaluate each experiment: the peak memory usage (column 'Mem') and running time (column 'Time') of UPPAAL, and the total number of states explored. Table 5.1 gives the comparison results for the scenario Direct Copy in parallel with the Simple Print and Table 5.2 shows the Direct Copy case in parallel with Process from Store. The first two columns in each table give the total number of instances processed for each task of an RTTS.

To combat state space explosion, we applied the sweep line method of UPPAAL [CKM01]. This method implies the definition of some progress measures that help to evaluate the progress in the state space exploration. Given a state s, all states reachable from s have a progress measure which is greater than or equal to the progress measure of s. In this way, certain states are deleted on-the-fly during state space generation, since the progress measures ensure that these states can never be reached again. In our models, parameter valuations (e.g. task instances) define the progress measures.

The state space obtained from the generated models is between 41% and 71% smaller than the one obtained from the handcrafted models. Therefore, the state

Table 5.1: Comparison handcrafted models (grey) vs. generated models for the direct copy (dc) || simple print (sp) case; O.M. - out of memory

#dc	#sp	Mem (KB)	Time (s)	Make-span(s)	States Explored
2	3	4500	0.50	23	1130
		5124	0.40	23	413
7	10	5480	1.60	71	10578
		5408	1.10	71	3050
35	50	12808	11.31	367	149926
		9184	13.51	367	48196
70	100	26568	27.92	737	433816
		18016	43.74	737	155491
334	500	598996	279.70	3585	6843592
		282220	898.98	3585	3038099
667	1000	2321768	1304.87	7166	25206064
		1076196	3702.10	7166	11704000
903	1355	4165896	1937.88	9705	45225661
		1962636	7165.40	9705	21272017
904	1356	O.M.	O.M.	O.M.	O.M.
		1964952	7173.70	9715	21302397
1460	1960	O.M.	O.M.	O.M.	O.M.
		4053164	17055.36	15117	44117751

space explosion problem emerges later in the analysis of the RTTS-based models, and we could analyze a higher number of task instances.

There are two causes that lead to the large difference in the sizes of the state spaces generated. Firstly, each resource was modeled in the handcrafted modeling approach by a separate automaton that comprises three locations: IDLE, RUNNING and RECOVERING (see Figure 2.9). The first two locations corresponded to locations L1 and L2 in the task automaton. The RECOVERING location models a recovery phase that some resources like Scanner require. However, this phase is redundant for majority of resources. Therefore, the recovery phase is modeled in the generated models as a separate task. The other cause comes from the tasks that claim more than one resource. In the generated models, one could easily model this multi-resource claim by checking the number of resource units available (see Figure 5.7). By contrast, in the handcrafted models, a multi-resource claim is modeled with the help of third party automata placed between the RTTS and resource automata, an extra third party automaton being added for each resource claimed. The extra automaton registers the claim to the resource automaton then waits for the resource automaton to become available. When it becomes available, it sends the request to the resource automaton. On completion of the processing,

Table 5.2: Comparison handcrafted models (grey) vs. generated models for the direct copy (dc) || process from store (pfs) case; O.M. - out of memory

#dc	#pfs	Mem (KB)	Time (s)	Make-span(s)	States Explored
1	2	4456	0.40	15	704
		5516	0.40	15	411
10	20	7540	4.10	114	47551
		6936	6.41	114	20118
25	50	21352	19.51	279	334606
		13984	40.14	279	135453
120	240	586172	384.66	1324	8255421
		244260	991.61	1324	3269408
240	480	2555392	1857.78	2644	33327861
		1007540	4245.12	2644	13162088
303	606	4077452	2419.45	3337	53223828
		1572420	7053.88	3337	21007415
304	608	O.M.	O.M.	O.M.	O.M.
		1582848	7157.23	3348	21146664
480	960	O.M.	O.M.	O.M.	O.M.
		4056016	20827.78	5284	52819448

it sends back an end event to the RTTS automaton.

Tables 5.1 and 5.2 also show up to a 61% decrease in the peak memory used by UPPAAL during the analysis. However, analysis of the generated models required more time. This was the price to pay for the parametric representation of these models, where a lot of details were encoded into functions. The evaluation of some of these functions (e.g. dep_met) required a lot of time due to the conditions or function calls that they contain.

5.7 Conclusions

PPOs are a simple extension of partial orders, but expressive enough to compactly represent large task graphs with finite repetitive behavior. We presented a translation from a subclass of PPOs to UPPAAL, together with a correctness proof that the transition system induced by a UPPAAL model is isomorphic to the configuration structure of a PPO.

This chapter introduces RTTSs, a general model for real-time embedded systems that uses PPOs for modeling applications. A distinguishing feature of RTTSs is the ability to express that the pace of a task depends on the number of resources that have been allocated to it. We described a translation to UPPAAL for a significant subclass of RTTSs.

Finally, we presented experiments which demonstrate that the UPPAAL models obtained through our translation are more tractable than handcrafted models of the same systems used in Chapter 2.

As explained in this chapter, when the applications (use cases) of a real-time embedded system design are described using PPOs, then we have a well-defined partial order structure on the corresponding events. Due to competition for resources and timing constraints, only a fragment of all the interleavings of this partial order will be possible in the full system model. Nevertheless, it will be interesting to see whether partial order reduction techniques [Pel98, McM92, HP07] will allow us to exploit the inherent structure of PPOs to alleviate the state space explosion problem when analyzing the full system model.

Another interesting topic for future work is to adapt the results of [HvdNV03] to the PPO settings. This approach reduces the complexity of scheduling problems by exploiting the finite repetitive structure of tasks: it reduces a scheduling problem to a problem containing a minimal number of identical repetitions, and after solving this smaller problem, the computed schedule is expanded to a schedule for the original, more complex scheduling problem.

The experiments that we described in Section 6 of this chapter are entirely non-parametric: only after the values of all timing parameters, CPU speeds, buffer sizes, etc have been fixed, we may compute the reachable states of our model using UPPAAL and compute performance metrics such as the makespan. It would be very interesting to explore the possibility of a parametric analysis in order to speed up the design-space exploration, along the lines that have been explored in [CPR08, SRL+11].

All our UPPAAL models that include the dynamic behavior of the Océ systems embody an approximation of the time elapsed between two consecutive changes in task pace. In the next chapter, we scrutinize the validity of the schedules obtained with these models. The aim is to check if these schedules can be followed when we compute the exact duration of each task instance. In case the schedules are invalid, we measure their robustness in scenarios in which some input parameters are slightly modified.

Chapter 6

Schedule Robustness Analysis

In this chapter, we scrutinize the validity (or implementation) of the schedules presented in the other chapters and obtained with UPPAAL. The durations of some task instances are over-approximated, which might make the schedules invalid under non-lazy scheduling assumptions when computing the exact values. If these schedules are proven invalid, we investigate their robustness by slightly varying some task durations. Another applicability of this chapter comes from the fact that the durations of task instances can be estimated, but (since they run on physical machines) not known exactly. Therefore, it is important to study the robustness of these schedules when some preconditions are modified.

6.1 Introduction

Many researchers in the field of real-time scheduling aim at finding optimal solutions under deterministic and static environmental conditions; static in the sense that all tasks are known from the beginning of the analysis and deterministic meaning that all task and machine parameters are constant and known in advance. However, schedules that satisfy these assumptions are often difficult to follow in a real-time environment which possesses an element of uncertainty [SG09].

Therefore, other researchers search for schedules that preserve a good performance when they are exposed to uncertainties. Uncertainty can be caused by different unforeseen events, such as machine breakdowns, arrival of urgent orders, or variations in task execution times. Two approaches are being distinguished in the literature on scheduling under uncertainty (see [VHL03], [SG09], [ALM+05]): proactive and reactive scheduling. In the *proactive* approach, the focus is on constructing off-line a base-line schedule which incorporates a degree of uncertainty anticipation that might appear during execution. One way of achieving this, and that is of great interest in our study, is by generating *robust schedules* that pre-

serve relatively good performance at runtime in the presence of uncertainties. In contrast to proactive scheduling, a given base-line schedule is not necessarily developed under the *reactive* approach. The aim here is to find optimal strategies to recover after the occurrence of disruptive events. Different combinations of the two policies have been studied, such as *proactive-reactive scheduling* where a base-line schedule is created in a proactive manner and deviations from the original schedule can be handled in a reactive manner.

Different methods have been proposed in the literature for obtaining robust schedules, which constitutes the main subject of this chapter. A great deal of research papers focus on robust schedules synthesis and provide measurements of the robustness level with respect to a given performance criterion that should be preserved (e.g. [DK95, Her99, BM02, PSCO04]).

In their study, Daniels and Kouvekis [DK95] analyze the problem of robust scheduling in a single-machine environment with independent tasks and uncertain processing times. The authors present two methods for obtaining robust schedules based on worst case analysis. The first finds the absolute robust schedule, which is the schedule that gives the minimum difference over all scenarios between the maximum and optimal valuations of a schedule. The second seeks the schedule that minimizes the worst-case percentage deviation from optimality over all scenarios for a given performance measure. Here a scenario is a unique set of processing times for tasks. The authors define a branch-and-bound algorithm on an equivalent form of the first measure where the performance criterion is the total flow time. Kouvelis et al. [KDV00] extend of this work for a two-machine environment with makespan (i.e. the total time in which a set of tasks are processed) as the performance criterion.

Kasperski [Kas05] studies the single machine sequencing problem with maximum lateness criterion under the assumptions that task processing times and due dates are imprecise and specified as intervals. The author presents a polynomial time algorithm based on Lawler's algorithm to find a robust schedule which minimizes the worst-case performance over all scenarios.

In the same way, Herrmann [Her99] proposes a 2-space genetic algorithm to solve minimax optimization problems. The algorithm is used for solving a parallel machine scheduling problem with uncertain processing times with the objective of finding a schedule that minimizes the worst-case makespan.

These papers, however, assume that all machines can process at most one task at the time and there are no dependencies between tasks. These assumptions allowed the decision space to be discretized without losing important cases. However, in our setting, task dependencies are an important characteristic and, furthermore, we also analyze machines that can process more than one task simultaneously. Besides, we reason about robustness of non-lazy schedules (i.e. all tasks start immediately after their resources claimed become available).

Bölöni and Marinescu [BM02] study the robustness of systems that contain dependent tasks where some component execution times might vary. Their ro-

bustness metric is based on the concept of critical path, which is a path through a graph such that, if any component along the path is late, then the makespan will also become longer. Given the probabilities of individual components to be late, one may compute the probability of any particular path in the schedule to be critical. Then, a schedule with fewer critical paths is considered to be more robust. Furthermore, the authors introduce the entropy of a schedule as an alternative for measuring the robustness of a schedule. The entropy is based on the probability of a path to become critical. The solutions presented of this study are, however, not applicable in our case since our system is not probabilistic.

Shi et al. [SJD06] also define two slack-based metrics to improve robustness of schedules over makespan. A slack is an extra time each activity is extended with such that a given schedule can absorb small perturbations induced by the environment. The first metric defines robustness as the inverse of relative schedule tardiness (the difference between the expected makespan and the actual makespan after perturbations). The second considers robustness as the inverse of the miss rate that is the number of schedules that exceed the expected makespan. The experimental results show that minimizing makespan and maximizing the slack are two opposite objectives but slack is effective for improving robustness.

Jensen [Jen01] defines two metrics based on N_k neighborhood to find robust and flexible schedules for job shop problems where some resources breakdown. The N_k neighborhood is the Hamming distance between two schedules where the task order on the same machine changes. The author compares the N_1 neighborhood (schedules where only two consecutive tasks on the same machine are interchanged) with slack based techniques. The results show that for moderate breakdown time, the neighborhood-based robustness measures outperform the slack-based robustness measure but for large breakdown times, the latter performs better.

Policella et al. [PSCO04, Pol05] treat the problem of generating robust scheduling solutions based on Partial Order Schedules (POS). A POS is defined as a set of solutions for the Resource Constrained Project Scheduling Problem with Generalized Precedence Relations (RCPSP/max) that is compactly represented by a temporal graph. RCPSP/max assumes dependencies between tasks and resources that can process more than one resource at the time. Since we do not explore the entire solution space for finding a robust schedule, we cannot apply this result for our case where the focus is on measuring the robustness of a given schedule.

The work of [Sol11] defines robust solutions against resource unavailability in auctions based on supermodels (for SAT) and supersolutions (for Constrained Programming). They have evaluated their solution with Yices, BSOLO (a pseudo-Boolean solver) and CPLEX (an integer programming solver). Their results show that the latter two solvers are more efficient in terms of time, solving larger input combinations.

In this chapter, we study first the validity of the schedules obtained with UPPAAL presented in the previous chapter. These schedules contain an approximation of the duration of task instances. The purpose of this analysis is to check if these

schedules can be followed when we use the accurate values for task instance duration under non-lazy scheduling. We call this process *schedule validation*. In case these schedules are proven invalid, we investigate their robustness by using the notion of (a,b) *supermodels* [GPR98] from SAT theory. Satisfiability (SAT) is the problem of finding a solution (i.e. a *model*) for a set of propositional formulas that contain Boolean variables formed using logical connectives (e.g. conjunction, disjunction and negation). An (a,b) *supermodel* is a model such that if we modify the values taken by the variables in a set of size at most a (breakage), another model can be obtained by modifying the values of the variables in a disjoint set of size at most b (repair). Both analyses are performed with a Satisfiability Modulo Theory (SMT) solver, called Z3 [dMB08, Z3]. Another applicability of this chapter comes from the fact that task instance duration can be estimated, but (since the task instances run on physical machines) not exactly. Therefore, it is important to study the robustness of these schedules when some preconditions are modified.

Most successful SAT solvers perform a *systematic search*, namely they represent a formula as a tree having as vertices Boolean variables, and two edges out of each vertex representing a true or false valuation of a variable. A *model* is a path from the root to a leaf that makes the formula satisfiable. Virtually all SAT solvers implement the DPLL algorithm [DLL62], which is a backtracking-based search algorithm with three main operations: decide, propagate and conflict resolution on formulas represented in conjuncture normal form (CNF). A CNF formula is a conjunction of clauses which, in turn, are disjunctions of literals. Literals are made out of atoms (i.e. Boolean variables) or their negation. The decide operation assigns a true or false value to unassigned atoms. The propagate action checks the consequences of a partial truth assignment using deduction rules. Finally, the conflict resolution step assigns a new truth value for some literals that make a formula unsatisfiable. If there is no truth value unexplored, then the formula is unsatisfiable.

Due to their progress in the last years, Satisfiability Modulo Theory (SMT) solvers (e.g. Z3, Yices, Barcelogic, CVC) have received a lot of attention. They extend the SAT problem with solvers for decidable theories of first-order logic. A theory contains a set of decision procedures that fix the interpretation of certain predicate and function symbols. Example of theories are: linear arithmetic, difference arithmetic, uninterpreted functions, bit-vectors or arrays. One framework for combining theory solvers is the Nelson-Oppen method [NO79].

Z3 is used for instance for dynamic test generation in Pex [GdHN+08], in program model checking tools such as BLAST [HJMS03] and SMV, and also in static program analysis [dMB08] and model-based testing [GKM+08].

The structure of this chapter is as follows. Next section formally defines the properties of the schedules obtained from a real-time task system which is used to represent our Océ system. Then, we present the structure of the SMT-LIB scripts generated for schedule validation and robustness analyses. Section 6.4 gives the

validation results and Section 6.5 presents a robustness analysis of the UPPAAL schedules based on SMT solving. At the end of this chapter, we draw conclusions and make recommendations for further research.

6.2 Schedule Definition

As described in the previous chapter, we represent an Océ system as a real-time task system $\mathcal{RTTS} = (\mathcal{A}, \mathcal{R}, \mathsf{cap}, \mathsf{cl}, \mathsf{h}, \mathsf{w}, \rho)$ in combination with a number of scheduling rules that prune away certain transitions and configurations in the semantics. In order to ensure that the real-time task system can be translated to UPPAAL, we require that $\mathcal{RTTS}$ and the scheduling rules satisfy a number of restrictions, listed in Section 5.5. In particular, we require that for each timed configuration that is allowed by the scheduling rules, the resource allocation is uniquely determined, through a function alloc, by the set of events that have occurred, that is, the configuration of the underlying PPO.

A path $\rho = (C_0, O_0, \theta_0) \xrightarrow{a_1} (C_1, O_1, \theta_1) \xrightarrow{a_2} (C_2, O_2, \theta_2) \ldots$ of the timed configuration structure $\mathcal{TC}(\mathcal{RTTS})$ is called *valid* if each timed configuration and each transition on the path is allowed by the scheduling rules. Path ρ is called *complete* if all the reachable events occur in it, that is, $\mathsf{rev}(\mathcal{A}) \subseteq \{a_1, a_2, \ldots\}$. To each complete path ρ, we can associate a unique function $\tau : \mathsf{rev}(\mathcal{A}) \to \mathbb{R}_{\geq 0}$ that assigns an occurrence time to each reachable event by adding up all time delays from the beginning of ρ until an event occurrence. We call a function $\tau : \mathsf{rev}(\mathcal{A}) \to \mathbb{R}_{\geq 0}$ a *schedule* of $\mathcal{RTTS}$ if τ is associated to some complete path of $\mathcal{RTTS}$. A schedule τ is *valid* if it is associated to a valid path. We further assume only valid schedules. In this section, we list a number of constraints that are satisfied by any valid schedule τ of a given real-time task system $\mathcal{RTTS}$.

We first need to define a few auxiliary functions. Recall that operations on resource allocation are defined by pointwise extension, see footnote 1 in Section 5.4.

Function $alloc^x_\tau$ gives the resource allocation for the configuration at time x:

$$alloc^x_\tau \quad = \quad \mathsf{alloc}(\{\alpha \in \mathsf{rev}(\mathcal{A}) \mid \tau(\alpha) \leq x\}).$$

Function $active_\tau$ returns the set of task instances that are active at a given time $x \in \mathbb{R}_{\geq 0}$ and that are using resources:

$$active_\tau(x) \quad = \quad \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \tau(\mathsf{start}(\beta)) \leq x < \tau(\mathsf{end}(\beta)) \wedge alloc^x_\tau(\beta) > \mathbf{0}\}.$$

Function $waiting_\tau$ returns the task instances that have been handed over resources, but have not started yet at a given time $x \in \mathbb{R}_{\geq 0}$:

$$waiting_\tau(x) \quad = \quad \{\beta \in \mathsf{ti}(\mathcal{A}) \mid \tau(\mathsf{start}(\beta)) > x \wedge alloc^x_\tau(\beta) > \mathbf{0}\}.$$

Then any schedule τ of an $\mathcal{RTTS}$ will satisfy the following constraints:

- **F1** *(Precedence order)*: If event α is an immediate predecessor of α', i.e., $\alpha \mapsto \alpha'$, then α occurs earlier or at the same time as α':

$$\alpha \mapsto \alpha' \quad \Rightarrow \quad \tau(\alpha) \leq \tau(\alpha')$$

- **F2** *(Task instance duration)*: If β is an instance of a task T that is non-preemptive and only claims static resources, then the duration of β is fixed:

$$\tau(\mathsf{end}(\beta)) \quad = \quad \tau(\mathsf{start}(\beta)) + \frac{\mathsf{w}(T)}{\rho(T, \mathsf{cl}(T))}$$

  In case a task instance uses dynamic resources or is preemptive, more specific constraints are needed to express the relationship between the occurrence time of the corresponding start and end events. In the Océ case study, USB task instances change their duration dynamically. More details about this will be given in the next section.

- **F3** *(Resource occupation)*: The number of units of any resource occupied at the start of any task instance β should not exceed the total number of units available:

$$\sum_{\beta' \in active_\tau(\tau(\mathsf{start}(\beta))) \cup waiting_\tau(\tau(\mathsf{start}(\beta)))} alloc_\tau^{\tau(\mathsf{start}(\beta))}(\beta') \leq \mathsf{cap}$$

  Resources that have one resource unit available are called *disjunctive*, whereas resources with more units available are called *cumulative*.

The *makespan* M of a *schedule* τ is defined as the difference between the occurrence times of the last and the first events:

$$M(\tau) \quad = \quad \max_{\alpha \in \mathsf{rev}(\mathcal{A})} \{\tau(\alpha)\} - \min_{\alpha \in \mathsf{rev}(\mathcal{A})} \{\tau(\alpha)\}.$$

Given a schedule τ of a real-time task system $\mathcal{RTTS}$ and a task instance $\beta \in \mathsf{ti}(\mathcal{A})$, function $est_\tau : \mathsf{ti}(\mathcal{A}) \to \mathbb{R}_{\geq 0}$ returns the *earliest starting time* of β and is equal to the moment when all immediate predecessors of the start event of β have occurred:

$$est_\tau(\beta) \quad = \quad \max_{\alpha \in \mathsf{rev}(\mathcal{A}) | \alpha \mapsto \mathsf{start}(\beta)} \tau(\alpha).$$

A schedule τ is *non-lazy* if, whenever the start event of an instance β of a non-preemptive task T occurs later than its immediate predecessors, then, for any time moment $x \in [est_\tau(\beta), \tau(\mathsf{start}(\beta)))$, insufficient resources are available to start β:

$$\textbf{F4 } \textit{(Non-laziness)} : x \in [est_\tau(\beta), \tau(\mathsf{start}(\beta))) \quad \Rightarrow \quad \rho(T, alloc_\tau^x(\beta)) = 0.$$

6.3 SMT-LIB Script Structure

We have written a small C# application which automatically generates SMT-LIB scripts from a UPPAAL schedule and an RTTS description. The scripts use the SMT-LIB language (version 1.2), which is a common language among the SMT solvers and are checked for satisfiability with Z3, version 4.0. Each script has the following structure:

```
1 (benchmark <script_name>
2 :extrafuns ((<jobId_taskId_instanceNo_eventType> Real))
3 :extrafuns ((<jobId_taskId_instanceNo_d> Real))
4 :assumption (<A1>)
5     ...
6 :assumption (<An>)
7 :formula (<F>)
8 )
```

The occurrence time of each event is declared as a constant of Real type using attribute *extrafuns* as exemplified above in the second line. The name of each constant follows the pattern jobId_taskId_instanceNo_eventType. Each task duration is also represented as a real-valued constant (Line 3). SMT-LIB version 1.2 allows two types of formulae: assumptions and formula. The difference between assumptions and formula is purely operational, related to the fact that many SMT solvers can process assumptions more efficiently if they are explicitly defined. Assumptions are not utilized in the validation analysis. The formula <F> contains the **F1-F4** constraints defined in Section 6.2.

The formula <F> also contains the order between concurrent task instances extracted from the UPPAAL schedule. These task instances have the property that they share disjunctive resources. Thereby, we define a new set of edges, called *disjunctive edges*, which impose an order between start events of task instances that share disjunctive resources. Let $\mathfrak{D}_\tau$ contain the set of disjunctive edges:

$$\begin{aligned}\mathfrak{D}_\tau = \{&(\mathsf{start}(\beta), \mathsf{start}(\beta')) \in \mathsf{ev}(\mathcal{A}) \times \mathsf{ev}(\mathcal{A}) \mid \mathsf{task}(\beta) \neq \mathsf{task}(\beta') \wedge \exists r \in \mathcal{R}\ s.t.\\ &\mathsf{cap}(r) = \mathsf{cl}(\mathsf{task}(\beta))(r) = \mathsf{cl}(\mathsf{task}(\beta'))(r) = 1 \wedge \tau(\mathsf{end}(\beta)) \leq \tau(\mathsf{start}(\beta'))\\ &\wedge \nexists \beta'' \in \mathsf{ev}(\mathcal{A})\ s.t.\ \mathsf{cl}(\mathsf{task}(\beta''))(r) = 1 \wedge \mathsf{start}(\beta) < \mathsf{start}(\beta'') < \mathsf{start}(\beta')\}\end{aligned}$$

We further use the schedule of Figure 6.1 for illustrating the structure of the SMT-LIB scripts produced for validating UPPAAL schedules. The white rectangles represent tasks instances of the direct copy (dc) RTTS (Figure 5.6), and the gray rectangles represent task instances of print from store (pfs) RTTS (Figure 5.4).

The set of disjunctive edges obtained for this schedule is:

$$\begin{aligned}\mathfrak{D}_\tau = \{&(pfs\_up\_1\_s, dc\_up\_1\_s), (dc\_up\_1\_s, pfs\_up\_2\_s),\\ &(pfs\_ip1\_1\_s, dc\_ip1\_1\_s), (dc\_ip1\_1\_s, pfs\_ip1\_2\_s),\\ &(pfs\_ip2\_1\_s, dc\_ip2\_1\_s), (dc\_ip2\_1\_s, pfs\_ip2\_2\_s)\}.\end{aligned}$$

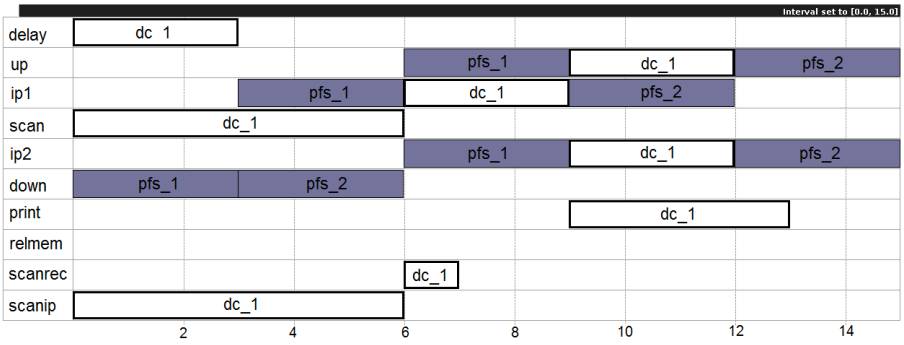


Figure 6.1: UPPAAL Schedule

The notation of the events in the above set is compound of: a prefix for the RTTS name, the resource claimed, a natural number that indicates the task instance, and a suffix *s* that marks a start event.

An SMT-LIB script incorporates the following parts:

- *Event initialization*

  In this phase, we declare a constant of Real type for the occurrence time of each event that appears in a UPPAAL schedule. For example, the next line defines the constant that records the occurrence time of the start event of the first scan task instance.

  ```
  :extrafuns ((dc_scan_1_s Real))
  ```

  Further, we assign constraints regarding the occurrence time of each end event. This step corresponds to rule **F2** in the previous section. All task instances except for the *down* and *up* are non-preemptive and use only static resources, therefore, they preserve the same pace after they start. Accordingly, we require that each constant that records the occurrence of an end event is equal to the constant that records the occurrence of its corresponding start event plus a delay that represents task instance duration, made of task work divided by task pace. For example, the next line presents constraint for the end event of first scan task instance that requires that this event must occur after 6 time units from the corresponding start event:

  ```
  (= dc_scan_1_e (+ dc_scan_1_s 6.0))
  ```

  For the duration of each instance of a USB task, we have declared a symbolic constant of Real type. Moreover, we assume that the duration of each USB task instance is composed of two parts: a part that represents the total amount of time in which the task instance shares the USB and another for the total amount of time when the task instance solely uses this bus. Further, we have included a constraint that the task work is equal to the sum of work

transfered at low pace and the work transfered at high pace. The work at low pace is compounded of parts when the USB is bidirectionally used. For counting the time when other task instances also use the USB in the opposite direction, we use of the following rule:

```
1  input:
2   t: usb task instance,
3   o_tasks: list of all task instances that use the USB in the
        opposite direction
4  output:
5   shared_dur : the total period in the duration of t in which
        the USB is shared
6  foreach (t' in o_tasks)
7     if (t'_s <= t_s)
8       if (t'_e <= t_e)
9         if (t'_e > t_s) shared_dur += t'_e - t_s fi
10      else shared_dur += t_e - t_s fi
11    else
12      if (t'_e <= t_e) shared_dur += t'_e - t'_s
13      else
14        if (t'_s < t_e) shared_dur += t_e - t'_s fi
15      fi
16    fi
17 endfor
```

Given a USB task instance t (Line 2 in the above listing), let o_tasks contain the task instances that use the USB in the opposite direction. If a concurrent task instance $t' \in o\_tasks$ is started earlier than t (Line 7) and t' ends in between the start and end of t (Lines 8-9), then t has shared the USB with t' for the time given by the difference between the occurrence of the end event of t' and start event of t (Line 9). If t' ends after t (Line 10), then the USB is shared for the entire duration of task t. Lines 12-16 analyze the other case: when the concurrent task instance t' starts later than t and t and t' might share the USB either for the entire duration of t' (Line 12) or until the end of t (Line 14).

The following listing shows the implementation of the constraint regarding the occurrence of the end event of the first down task instance given by variable pfs_down_1_e. Variable pfs_down_1_d_shared gives the total amount of time in which the USB is shared and variable pfs_down_1_d_rest gives the total amount of time when the first down task instance uses solely the bus. Line 48 requires that task work (i.e. 12 units) is equal to the sum of work transfered at low pace (i.e. 3Mbps) and the work transfered at high pace (i.e. 4Mbps). Line 49 asserts that the total time in which the USB is shared is compounded of parts when the USB is bidirectionally used, which

corresponds to the rule described in the previous listing (see Lines 6-16).

```
44 (= pfs_down_1_e (+ pfs_down_1_s pfs_down_1_d))
45 (>= pfs_down_1_d_shared 0.0)
46 (>= pfs_down_1_d_rest 0.0)
47 (= pfs_down_1_d (+ pfs_down_1_d_shared pfs_down_1_d_rest))
48 (= (+ (* 3.0 pfs_down_1_d_shared) (* 4.0 pfs_down_1_d_rest))
       12.0)
49 (= pfs_down_1_d_shared (+ (ite (<= pfs_up_1_s pfs_down_1_s) (
       ite (<= pfs_up_1_e pfs_down_1_e) (ite (> pfs_up_1_e
       pfs_down_1_s) (- pfs_up_1_e pfs_down_1_s) 0.0) (-
       pfs_down_1_e pfs_down_1_s)) (ite (<= pfs_up_1_e
       pfs_down_1_e) (- pfs_up_1_e pfs_up_1_s) (ite (< pfs_up_1_s
       pfs_down_1_e) (- pfs_down_1_e pfs_up_1_s) 0.0)))(ite (<=
       dc_up_1_s pfs_down_1_s) (ite (<= dc_up_1_e pfs_down_1_e) (
       ite (> dc_up_1_e pfs_down_1_s) (- dc_up_1_e pfs_down_1_s)
       0.0) (- pfs_down_1_e pfs_down_1_s)) (ite (<= dc_up_1_e
       pfs_down_1_e) (- dc_up_1_e dc_up_1_s) (ite (< dc_up_1_s
       pfs_down_1_e) (- pfs_down_1_e dc_up_1_s) 0.0)))(ite (<=
       pfs_up_2_s pfs_down_1_s) (ite (<= pfs_up_2_e pfs_down_1_e)
       (ite (> pfs_up_2_e pfs_down_1_s) (- pfs_up_2_e pfs_down_1_s
       ) 0.0) (- pfs_down_1_e pfs_down_1_s)) (ite (<= pfs_up_2_e
       pfs_down_1_e) (- pfs_up_2_e pfs_up_2_s) (ite (< pfs_up_2_s
       pfs_down_1_e) (- pfs_down_1_e pfs_up_2_s) 0.0)))))
```

- *Precedence order*
  This phase corresponds to rule **F1** in the previous section. Task instances that have no predecessor must start at time zero. For ensuring this, the constants that correspond to the occurrence of their start events must be equal to zero (Lines 98 and 101 listed below). In case of more task instances which can start at time zero and claim the same disjunctive resource, one of them must start at time zero. Otherwise, a task instance may start when all its predecessors have occurred (e.g. Line 99). In contrast with the UPPAAL models of Section 5.5, in the SMT-LIB scripts we have unfolded each PPO. In this way, we could for example specify that the start of the second pfs down instance occurs after the end of the first task instance (Line 102). In addition, in this phase we have included constraints given by the set of disjunctive edges $\mathfrak{D}_\tau$. For example, Lines 105 and 106 assert the order between the task instances that use the IP1 resource.

```
98 (= dc_scan_1_s 0.0)
99 (<= dc_scan_1_s dc_scanip_1_s)
100 (= dc_scanip_1_s dc_delay_1_s)
101 (= pfs_down_1_s 0.0)
```

```
(<= pfs_down_1_e pfs_down_2_s)
(<= pfs_down_1_e pfs_down_2_s)
(<= pfs_down_1_e pfs_ip1_1_s)
(<= pfs_ip1_1_s dc_ip1_1_s)
(<= dc_ip1_1_s pfs_ip1_2_s)
```

- *Resource occupation*
  This phase corresponds to rule **F3** from the list of constraints on schedules given in the previous section. If a task claims at least one resource, we require that all resources claimed are available at the start of any of its instances. The implementation of this rule and the next one is challenging due to the large number of task instances necessary to consider which make the resulting formulas extremely large. To avoid this effect, we inspected the UPPAAL schedules and took the active task instances at the start of each task instance. Then, we only inquired about the status of these task instances, their previous task instances and their next task instances. The following listing refers to the memory check at the start of the first scan task instance. Since there is no other task instance that uses the Scanner, we do not inquire about its availability. However, we require that the sum of the number of memory units occupied by the possible concurrent tasks is less than what the scan task requires.

```
(=> (>= dc_scan_1_s 0.0)(<= (+ (ite (and (<= pfs_down_1_s
    dc_scan_1_s) (> pfs_down_1_e dc_scan_1_s)) 24 0) (ite (and
    (<= pfs_down_2_s dc_scan_1_s) (> pfs_down_2_e dc_scan_1_s))
     24 0) (ite (and (> pfs_ip1_1_s dc_scan_1_s) (<=
    pfs_down_1_e dc_scan_1_s)) 24 0) (ite (and (<= pfs_ip1_1_s
    dc_scan_1_s) (> pfs_ip1_1_e dc_scan_1_s)) 24 0) (ite (and
    (> pfs_ip1_2_s dc_scan_1_s) (<= pfs_down_2_e dc_scan_1_s))
    24 0) (ite (and (<= pfs_ip1_2_s dc_scan_1_s) (> pfs_ip1_2_e
     dc_scan_1_s)) 24 0) (ite (and (> pfs_ip2_1_s dc_scan_1_s)
    (<= pfs_ip1_1_e dc_scan_1_s)) 24 0) (ite (and (<=
    pfs_ip2_1_s dc_scan_1_s) (> pfs_ip2_1_e dc_scan_1_s)) 12 0)
     (ite (and (> pfs_ip2_2_s dc_scan_1_s) (<= pfs_ip1_2_e
    dc_scan_1_s)) 24 0) (ite (and (<= pfs_ip2_2_s dc_scan_1_s)
    (> pfs_ip2_2_e dc_scan_1_s)) 12 0) (ite (and (> pfs_up_1_s
    dc_scan_1_s) (<= pfs_ip2_1_s dc_scan_1_s)) 12 0) (ite (and
    (<= pfs_up_1_s dc_scan_1_s) (> pfs_up_1_e dc_scan_1_s)) 12
    0) (ite (and (> pfs_up_2_s dc_scan_1_s) (<= pfs_ip2_2_s
    dc_scan_1_s)) 12 0) (ite (and (<= pfs_up_2_s dc_scan_1_s)
    (> pfs_up_2_e dc_scan_1_s)) 12 0)) 48))
```

One if-clause like the one below checks whether a task instance, namely $task\_inst\_A$ is active at the start of a task instance (i.e. $task\_inst\_B$), in

which case the memory occupation is positive (i.e. x MB), otherwise is zero.

```
(+ (ite (and (<= task_inst_A_s task_inst_B_s) (> task_inst_A_e
     task_inst_B_s)) x 0)
```

- *Non-lazy scheduling*
  This phase corresponds to rule **F4** of Section 6.2. The implementation of this rule is similar with the implementation of the previous rule in the sense that we need to add up the resource units occupied either at the earlier starting time or at any end event of a concurrent task instance and assert that either there are not enough units available for the task instance to start or it should start at that moment. If there is no competing task (see Line 147), the task must start at its earlier starting time.

```
147 (= dc_scan_1_s dc_scanip_1_s)
```

Line 156 below assures that if the second instance of task pfs_down happens after its predecessor (i.e. the first instance of task pfs_down), then there was not enough memory to accommodate the task instance at that moment. As mentioned above, we also check whether a task instance could start at the end of a concurrent task instance. Note that concurrent task instances are not only those stored in $\mathfrak{D}_\tau$, but any task instance that have resources in common with the task instance in question. Line 158 checks whether the start of the second instance of task pfs_down can occur at the end of the first instance instance of dc_ip2 because these instances share memory.

```
156 (=> (< pfs_down_1_e pfs_down_2_s)(> (+ (ite (and (>
    pfs_ip1_1_s pfs_down_1_e) (<= pfs_down_1_e pfs_down_1_e))
    24 0) (ite (and (<= pfs_ip1_1_s pfs_down_1_e) (>
    pfs_ip1_1_e pfs_down_1_e)) 24 0) (ite (and (> dc_ip1_1_s
    pfs_down_1_e) (<= dc_scanip_1_s pfs_down_1_e)) 48 0) (ite (
    and (<= dc_ip1_1_s pfs_down_1_e) (> dc_ip1_1_e pfs_down_1_e
    )) 48 0) (ite (and (> pfs_ip2_1_s pfs_down_1_e) (<=
    pfs_ip1_1_e pfs_down_1_e)) 24 0) (ite (and (<= pfs_ip2_1_s
    pfs_down_1_e) (> pfs_ip2_1_e pfs_down_1_e)) 12 0) (ite (and
     (> dc_ip2_1_s pfs_down_1_e) (<= dc_ip1_1_e pfs_down_1_e))
    48 0) (ite (and (<= dc_ip2_1_s pfs_down_1_e) (> dc_ip2_1_e
    pfs_down_1_e)) 48 0) (ite (and (> pfs_up_1_s pfs_down_1_e)
    (<= pfs_ip2_1_s pfs_down_1_e)) 12 0) (ite (and (<=
    pfs_up_1_s pfs_down_1_e) (> pfs_up_1_e pfs_down_1_e)) 12 0)
     (ite (and (> dc_relmem_1_s pfs_down_1_e) (<= dc_ip2_1_e
    pfs_down_1_e)) 12 0) (ite (and (<= dc_relmem_1_s
    pfs_down_1_e) (> dc_relmem_1_e pfs_down_1_e)) 12 0)) 72))
157
```

```
158 (=> (and (> dc_ip2_1_e pfs_down_1_e) (<= dc_ip2_1_e
    pfs_down_2_s)) (or(> (+ (ite (and (> pfs_ip1_1_s dc_ip2_1_e
    ) (<= pfs_down_1_e dc_ip2_1_e)) 24 0) (ite (and (<=
    pfs_ip1_1_s dc_ip2_1_e) (> pfs_ip1_1_e dc_ip2_1_e)) 24 0) (
    ite (and (> dc_ip1_1_s dc_ip2_1_e) (<= dc_scanip_1_s
    dc_ip2_1_e)) 48 0) (ite (and (<= dc_ip1_1_s dc_ip2_1_e) (>
    dc_ip1_1_e dc_ip2_1_e)) 48 0) (ite (and (> pfs_ip2_1_s
    dc_ip2_1_e) (<= pfs_ip1_1_e dc_ip2_1_e)) 24 0) (ite (and
    (<= pfs_ip2_1_s dc_ip2_1_e) (> pfs_ip2_1_e dc_ip2_1_e)) 12
    0) (ite (and (> dc_ip2_1_s dc_ip2_1_e) (<= dc_ip1_1_e
    dc_ip2_1_e)) 48 0) (ite (and (<= dc_ip2_1_s dc_ip2_1_e) (>
    dc_ip2_1_e dc_ip2_1_e)) 48 0) (ite (and (> pfs_up_1_s
    dc_ip2_1_e) (<= pfs_ip2_1_s dc_ip2_1_e)) 12 0) (ite (and
    (<= pfs_up_1_s dc_ip2_1_e) (> pfs_up_1_e dc_ip2_1_e)) 12 0)
     (ite (and (> dc_relmem_1_s dc_ip2_1_e) (<= dc_ip2_1_e
    dc_ip2_1_e)) 12 0) (ite (and (<= dc_relmem_1_s dc_ip2_1_e)
    (> dc_relmem_1_e dc_ip2_1_e)) 12 0)) 72)(= pfs_down_2_s
    dc_ip2_1_e)))
```

6.4 Validation Results

Figure 6.2 presents the input and output of Z3 for the UPPAAL schedule validation analysis. From a UPPAAL schedule we extract the set of disjunctive edges and from the RTTS file we extract the **F1-F4** constraints. Given these inputs we generate an SMT-LIB script that is checked with Z3. Z3 outputs an answer of the form 'SAT' or 'UNSAT'. If Z3 returns 'SAT', then the model obtained encodes a schedule which preserves the same ordering between task instances as in the UPPAAL input schedule, but the occurrence times of events may be left/right shifted. If the result is 'UNSAT', then some of the edges of $\mathfrak{D}_\tau$ cannot hold anymore when the exact computation of task instance durations is used in conjunction with non-lazy scheduling.

In this section, we present the validation results of some schedules discussed in Section 5.6. The scenario which we analyze here is the direct copy (dc) in parallel with print from store (pfs) RTTSs, but the conclusions also hold for the other scenario. For the analysis we have used Z3 version 4.0, 64-bit on a Sun Fire X4440 server with 16 cores (AMD Opteron 8356, 2.3GHz) and 128 Gb of DDR2 RAM.

Table 6.1 shows that we have validated some combinations of dc and pfs jobs of Table 5.2. The table also shows that we have obtained the same makespan for the schedules obtained with Z3 as for the UPPAAL schedules. We explored a few other combinations but the amount of time and memory significantly grew. However, we do not need more experiments because a closer look at the USB usage reveals that the upload and download tasks synchronize perfectly (see Figure 6.3) due

Table 6.1: UPPAAL schedule validation - synchronous USB events.

#dc	#pfs	Z3 Mem(KB)	Z3 Time(s)	Z3 Makespan(s)	UPPAAL Makespan(s)
1	2	35728	0.08	15	15
5	10	763152	110.42	59	59
10	20	1035028	1098.25	114	114
15	30	3828858	8894.95	169	169

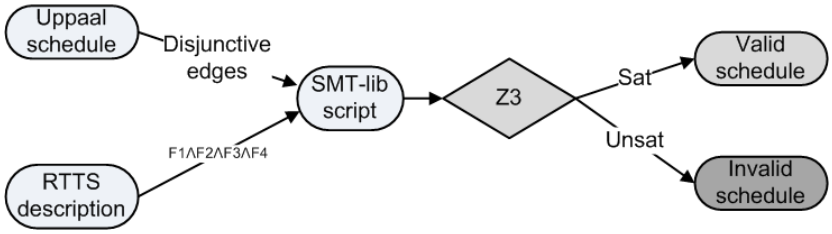


Figure 6.2: Diagram of the validation analysis of UPPAAL schedules.

to the fact that the amount of work for uploads and downloads is the same and there is no delay before the first pfs_down and dc_scan instances. It remains to explore in the next section the implications of different durations for the up and down tasks.

6.5 Schedule Robustness

If the exact valuation of the USB task instances differs from the valuation obtained with UPPAAL, it is highly probable that the order between task instances changes. Therefore, some disjunctive edges of $\mathfrak{D}_\tau$ do not hold anymore, meaning that the schedules are invalid under a non-lazy scheduling policy. As a consequence, robustness is a more natural property to prove in this case instead of validation. We define *robustness* as a property of a schedule that guarantees that the schedule can be (partially) followed even though some uncertainty might occur at runtime (e.g. some task duration become shorter/longer).

For evaluating the robustness of a schedule, we employ the notion of (a, b) supermodel introduced in [GPR98] for SAT formulas. It refers to models where if a

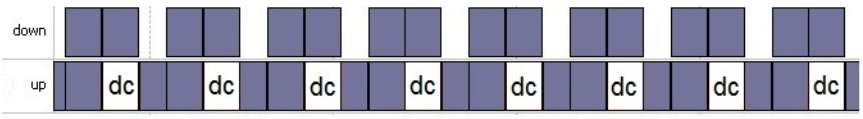


Figure 6.3: Pattern of the USB usage.

variables change their values, at most b different variables need reallocation. Similarly, in constraint programming the notion of super-solution [HHW04] has been devised. The difficulty in implementing this definition is in finding the right parameters to evaluate the consequences brought by changes in task instance duration. Since in our setting, memory is shared between all tasks, by changing the duration of a single task instance, the events that occur afterwards have unavoidably different timestamps, which would be misleading to measure. However, more appropriate is to compute the maximum number of unsatisfiable disjunctive edges. For this, we use the absolute Hamming distance [Jen01], which shows the difference between two schedules. In our setting, this distance represents the maximum number of disjunctive edges that are different between the original UPPAAL schedule and schedules with slightly modified USB task durations.

Given two schedules τ and τ' which fulfill the **F1-F4** rules, the *absolute Hamming distance* $H(\tau, \tau')$ is defined as follows:

$$H(\tau, \tau') = \frac{1}{2} \sum_{\beta, \beta' \in \mathsf{ti}(\mathcal{A})} [(\beta, \beta') \in \mathfrak{D}_\tau \oplus (\beta, \beta') \in \mathfrak{D}_{\tau'}],$$

where $\oplus$ represents the logical operator exclusive or. The term under summation is 1 if (β, β') is in $\mathfrak{D}_\tau$ and not in $\mathfrak{D}_{\tau'}$ or vice versa. The absolute Hamming distance basically computes the size of the complement of set $\mathfrak{D}_\tau$ with respect to the set $\mathfrak{D}_{\tau'}$.

Similar to [Jen01], we define the notion of $\mathcal{N}_k(\tau)$ neighbor of a schedule τ as the set of schedules τ' that fulfill the **F1-F4** constraints and have an absolute Hamming distance less or equal to k:

$$\mathcal{N}_k(\tau) = \{\tau' \vDash F1 \wedge F2 \wedge F3 \wedge F4 \mid H(\tau, \tau') \leq k\}.$$

We define an *(a, b)-supermodel* as a model that describes a schedule τ where if a task instances change their duration, all resulting schedules are $\mathcal{N}_b(\tau)$ neighbors of τ.

We evaluate the robustness of a UPPAAL schedule by finding its $\mathcal{N}_k$ neighbors in settings where we vary the work of different USB tasks. If k is large, then it is better to generate new schedules with the new values for the USB work than to use the old UPPAAL schedules, in which case we conclude that the schedules are not robust.

The (a,b)-supermodel problem is not a satisfiability problem anymore, but an optimization problem known in the literature as the partial MaxSMT problem. This problem contains two types of constraints: hard and soft. The solution for the partial MaxSAT/MaxSMT problem is the one which provides the maximum number of soft constraints satisfied when all hard constraints are obligatorily satisfied.

For solving the partial MaxSMT problem, we have used the MaxSAT module provided in the Z3 distribution. This module contains a simple MaxSAT solver (in C) on top of the Z3 API that parses SMT-LIB scripts, version 1.2. Each script contained the **F1-F4** rules of the original UPPAAL schedules. The rules regarding the $\mathfrak{D}_\tau$ set were negated and specified using assumption attributes like in the listing below. Therefore, we utilized assumptions for encoding soft constraints and a formula for hard constraints. The two assumptions in the listing below state that the task instance dc_ip1_1 should not be executed in between the first and second pfs_ip1 task instances.

```
:assumption (not (<= pfs_ip1_1_s dc_ip1_1_s))
:assumption (not (<= dc_ip1_1_s pfs_ip1_2_s))
```

In order to obtain the $\mathcal{N}_b(\tau)$ neighbors of τ, we need to solve the following optimization problem:

Minimize	$\mathfrak{D}_\tau$
Subject to	$F1 \wedge F2 \wedge F3 \wedge F4 \wedge$ different task works

The MaxSAT module implements two algorithms for solving an MaxSMT problem. The algorithm we have employed is based on the Fu & Malik procedure [FM06], which uses unsat-core extraction. Unsat-core represents a subset of the set of all constraints that the solver has determined to be unsatisfiable. The Fu & Malik procedure consists in iteratively calling a solver on a working formula. If the formula is unsatisfiable, the solver provides an unsat-core. Additional variables are further produced for each soft constraint that appears in the unsat-core. The new working formula will consist in adding the new variables to the old formula with the extra restriction that exactly one of the new variables should be true. This procedure is repeated until the solver returns satisfiable. Basically, at each iteration one soft constraint of the unsat-core is blocked. The total number of iterations subtracted from the total number of soft constraints represents the solution for the MaxSMT problem.

Tables 6.2 - 6.4 show how changes in the work of different USB tasks (i.e. Download, Upload of the dc RTTS and Upload of the pfs RTTS) reflect in the number of broken disjunctive edges. Symbol a shows the number of task instances with different work than in the original UPPAAL schedules. Column %work represents the percentage with which the work is decreased or increased. Column $|\mathfrak{D}_\tau|$ indicates the size of the set that contains disjunctive edges. Column b shows the maximum number of disjunctive edges that do not hold anymore. Column M_b gives the makespan of the schedule for the case when b disjunctive edges are violated. The last two columns show the peak memory and time that Z3 took for the analysis.

We can observe that only when the work of the Download task is positively increased, b is relatively small and we obtain the lowest number of unsatisfied dis-

Table 6.2: Robustness analysis results when varying the work of a Download task instances. Column b indicates the number of disjunctive edges damaged.

#dc	a=#pfs	%work	$\lvert\mathfrak{D}_\tau\rvert$	b	M_b	Memory(KB)	Time(s)
1	2	-0.083	6	3	15.75	35872	0.20
5	10		30	15	57.93	162808	203.19
10	20		60	30	110.02	1132596	2826.65
15	30		90	45	162.10	17945376	44554.0
1	2	0.083	6	3	15.16	36192	0.14
5	10		30	3	60.5	183800	312.76
10	20		60	3	117.16	1244220	5858.01
15	30		90	3	173.83	16794832	74566.0
1	2	0.166	6	3	15.33	36384	0.16
5	10		30	3	62	172868	206.99
10	20		60	3	120.33	1290752	8006.90
15	30		90	3	178.66	15849488	39421.0
1	2	0.25	6	3	15.5	36240	0.15
5	10		30	3	63.5	192628	411.45
10	20		60	3	123.5	1305668	8195.36
15	30		90	3	183.5	15516576	36795.0

junctive edges. This allows us to conclude that the corresponding Uppaal schedules are robust under these circumstances. Basically, in each schedule obtained when b disjunctive edges are broken, three pairs of concurrent task instances have swapped the order and the rest of task instances are right shifted. In the other cases, each second task instance switches the order with the next concurrent task instance, therefore half of the disjunctive edges cannot be preserved. With respect to the makespan, the least sparse results have been obtained by varying the work of the task dc Upload, whose instances are less numerous than the pfs task instances. In addition, the experiments show that changes in the work of tasks which are at the beginning of a use case have a larger impact on the makespan than changes in the work of tasks that occur at the end of a use case. This effect can be observed in case the work of the Download task is modified where the difference between the makespans obtained and the makespans of the original Uppaal schedules is much larger than for the other two cases.

6.6 Conclusions

In this chapter, we examined the validity of the non-lazy Uppaal schedules presented in the previous chapter. These schedules contain an approximation of the time elapsed since the latest change in a task pace. We initially aimed at following the exact ordering given by these schedules and we saw that for the experiments

Table 6.3: Robustness analysis results when varying the work of a pfs Upload task instances. Column b indicates the number of disjunctive edges damaged.

#dc	a=#pfs	%work	$\lvert\mathfrak{D}_\tau\rvert$	b	M_b	Memory(KB)	Time(s)
1	2	-0.083	6	3	16	36256	0.17
5	10		30	15	58.41	177640	393.86
10	20		60	30	111	1269476	8741.02
15	30		90	45	164	16816272	68206
1	2	0.083	6	3	16	36240	0.18
5	10		30	15	61.16	164884	363.49
10	20		60	30	116.58	1250508	9875.81
15	30		90	45	172	16940912	75795
1	2	0.166	6	3	16	36160	0.15
5	10		30	15	62.73	168212	324.90
10	20		60	30	119.91	1255000	9611.16
15	30		90	45	176.89	17478496	82432
1	2	0.25	6	3	16.5	36272	0.23
5	10		30	15	64.5	187920	384.04
10	20		60	30	123.25	1251912	10201.40
15	30		90	45	181.65	17037920	75891

presented in the previous chapter, where the works of the USB up and down tasks are equal, these schedules are valid. However, when some works are modified, the Uppaal schedules cannot be precisely followed. The next topic addressed was to evaluate the robustness of the Uppaal schedules when varying the work of USB tasks, these tasks being the ones for which we have used approximations. This can be also seen as a validation step of the case when the USB task durations are imprecise in the Uppaal schedules. Schedule robustness was measured based on the notion of (a,b)-supermodel from the SAT literature and the absolute Hamming distance, all experiments being performed with Z3, an SMT solver. The analysis has shown cases when the new schedules obtained by varying the work of the USB tasks did not significantly differ from the original Uppaal schedules.

Even though the amount of time and memory Z3 required for solving the satisfiability and MaxSMT problems addressed here significantly increased when we evaluated large number of jobs, SMT solvers showed a lot of potential for real-time scheduling. It would be interesting as future step to further improve on the structure of the SMT-LIB scripts. However, the size of our robustness problem was complex due to the precedence order, claim and non-lazy rules explicitly implemented in the scripts. SMT solvers, though, assume that only a small part from large formulas is relevant for establishing satisfiability [dMB09].

As future research, we can embark on proving robustness of the schedules presented in Chapter 4 that contain most of the Océ controller scheduling rules.

Table 6.4: Robustness analysis results when varying the work of a dc Upload task instances. Column b indicates the number of disjunctive edges damaged.

a=#dc	pfs	%work	$\lvert\mathfrak{D}_\tau\rvert$	b	M_b	Memory(Mb)	Time(s)
1	2	-0.083	6	3	15	36432	0.17
5	10		30	15	59.29	174076	314.08
10	20		60	30	112.72	1257636	8409.46
15	30		90	45	166.25	17897408	90735
1	2	0.083	6	3	15.25	35952	0.12
5	10		30	15	59.81	177608	281.44
10	20		60	30	113.87	1230944	7757.67
15	30		90	45	167.93	16219872	66475
1	2	0.166	6	3	15.5	35904	0.17
5	10		30	15	60.12	167216	241.74
10	20		60	30	114.5	1270384	9016.98
15	30		90	45	168.87	16688736	67633
1	2	0.25	6	3	15.75	35872	0.17
5	10		30	15	60.43	159600	239.12
10	20		60	30	115.12	1233796	9061.69
15	30		90	45	169.81	16012656	58462

For the robustness analysis of schedules with large number of jobs, we can check if by slicing the scheduling problem into smaller problems and iteratively add more jobs, we obtain the same results. The idea is that for each slice, the soft assumptions of the previous slice are now hard assumptions and Z3 has to reason only about the additional soft assumptions that describe the new slice. For large number of jobs, this can be less costly to evaluate.

Chapter 7

Conclusions

In this thesis, we have focused on closing the gap between model-based performance evaluation tools and industrial system-level design problems. More concretely, we have explored the use of UPPAAL, a timed automata model checker, as the principal means for modeling and analyzing the datapath module of Océ printers. The architecture of such a module contains resources that manifest a dynamic behavior, their speed being increased or decreased after they commence processing a job. Moreover, several controller scheduling rules have to be considered when modeling this module.

For modeling the Océ datapath module, we created two models: one abstract and one refined. Both representations contained dynamic resources but in the refined model the majority of resources behaved dynamically and we incorporated majority of the scheduling rules of an Océ printer controller. The analysis of both models was targeted for a large number of jobs that pushed UPPAAL to its limits. In fact, it was not easy to strike a balance between finding the relevant details and keeping a reasonable size of the state space generated for these models. The amount of details and the large number of jobs we had to consider for a realistic analysis produced a great deal of interleavings between independent processes. Due to this, we often encountered state space explosion during verification. In this sense, we had to make use of several methods provided by UPPAAL to reduce the size of the state space such as: impose priorities between processes and channels, make urgent the majority of channels, and employ progress measures.

During the Octopus project, which provided the context for this thesis, we have faced the lack of a common and simple medium for the communication with the Océ engineers and avoiding modeling errors with the tools used (i.e. CPN Tools, UPPAAL and SDF3). The problem was that the models were constructed using formalisms that were not accessible to the design engineers and were difficult to ignore when it came to interpret analysis results. This resulted in a new toolset, called the Octopus toolset [Oct11, BVBG$^+$10], with the purpose of model-driven design-space exploration. In this thesis, we define a representation for real-time

task graphs based on parametrized partial orders used in the first version of the toolset. We also present a translation of this representation into UPPAAL and use it for redesigning our abstract UPPAAL model. The experiments showed that the UPPAAL models generated based on this representation were more tractable comparing to the more intuitive modeling style used in the abstract model. As future work, we can model and analyze the refined model with this representation.

The thesis also aims to make realistic assumptions about the behavior of the environment. Often, scheduling results cannot be exactly implemented in real machines because they do not assume uncertain environmental conditions. We reserve two chapters on this subject where we assume scenarios in which an infinite stream of jobs is interrupted by jobs with uncertain arrival times. In one chapter we have used UPPAAL-TIGA for schedule synthesis, UPPAAL-TIGA being an extension of UPPAAL solving timed games. In the other chapter we have used UPPAAL and the model constructed here is a refinement of the abstract model by including printer controller scheduling rules. The conclusion of the analysis with UPPAAL-TIGA is that it is currently not possible to use the tool directly for the generation of control software for our datapath case study: the strategies produced by UPPAAL-TIGA (albeit memoryless) are really big and contain thousands of rules. The analysis with UPPAAL provided a better resource priority used by one of the scheduling rules. However, although essentially the behavior of the model was fully specified with all scheduling rules incorporated, the resulting UPPAAL model was not fully deterministic (and suffers from state space explosion) due to the interleaving of internal actions of various resources.

For modeling dynamic resources, we computed the amount of time and work left after a change in the speed of a resource. Time is recorded by clocks, which are variables of type Real, and clock variables are not allowed in expressions. As a result, we under-approximated the value of the clocks that recorded the time spent between two consecutive changes in resource speed to the nearest integers. Consequently, we questioned the validity of the schedules obtained from the abstract UPPAAL model, which contained such timing approximations. For this we used Z3, a state-of-the-art SMT solver. Furthermore, we have evaluated the robustness of these schedules using the notion of supermodel from the SAT theory and represented it in Z3 as a partial MaxSMT problem.

Z3 has showed a lot of potential for the validation and robustness problems emerged from the UPPAAL analysis but it is not yet scalable. In general, SMT solvers are more suitable for real-time task systems that do not contain so many precedence constraints regarding task ordering like the ones examined here [dMB09]. As future work, we plan to search for a more efficient way of representing the UPPAAL schedules in the SMT-LIB approach to reduce the long running times.

To conclude, the UPPAAL language was proven expressive enough to describe all characteristics encountered in datapath design. In addition, even though it required some expertise to model an Océ datapath efficiently, UPPAAL has been shown valuable for the comparison of various design options.

In spite of an extensive usage of the Océ datapath case study throughout the thesis, we expect that many of the conclusions of this thesis can be applied to a much larger class of system-level design problems. In addition, the real-time task system representation proposed here can be adopted in many scheduling analysis problems using UPPAAL. This would not only benefit from an accurate analysis, but will also support future research in this area.

Bibliography

[AAG+07] Y. Abdeddaïm, E. Asarin, M. Gallien, F. Ingrand, C. Lesire, and M. Sighireanu. Planning Robust Temporal Plans: A Comparison Between CBTP and TGA Approaches. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 2–9. AAAI, 2007.

[AAM06] Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *Theoretical Computer Science*, 354(2):272–300, 2006.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AD94] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[AHI+09] I. AlAttili, F. Houben, G. Igna, S. Michels, F. Zhu, and F. W. Vaandrager. Adaptive Scheduling of Data Paths using Uppaal Tiga. In *Workshop on Quantitative Formal Methods: Theory and Applications (QFM)*, volume 13 of *EPTCS*, pages 1–11, 2009.

[AKM03] Y. Abdeddaïm, A. Kerbaa, and O. Maler. Task Graph Scheduling Using Timed Automata. In *International Parallel and Distributed Processing Symposium (IPDPS)*, page 237. IEEE Computer Society, 2003.

[ALM+05] H. Aytug, M. A. Lawley, K. Mckay, S. Mohan, and R. Uzsoy. Executing production schedules in the face of uncertainties: A review and some future directions. *European Journal of Operational Research*, 161(1):86–110, 2005.

[AME07] AMETIST. Final Project Report, August 2007. Deliverable from the European project IST-2001-35304 Advanced Methods for Timed Systems (AMETIST), http://www.cs.ru.nl/F.Vaandrager/AMETIST/final.pdf.

[BBHM05] G. Behrmann, E. Brinksma, M. Hendriks, and A. Mader. Production Scheduling by Reachability Analysis - A Case Study. In *International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2005.

[BCD+07] G. Behrmann, A. Cougnard, A. David, E. Fleury, K. Larsen, and D. Lime. Uppaal Tiga User-manual. `http://www.cs.aau.dk/~adavid/tiga/manual.pdf`, 2007.

[BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara. *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Press, 1997.

[BDL04] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on Uppaal. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems (SFM-RT)*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[BDL+06] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *International Conference on the Quantitative Evaluation of SysTems (QEST)*, pages 125–126. IEEE Computer Society, 2006.

[BGVZ11] J. Berendsen, B. Gebremichael, F. W. Vaandrager, and M. Zhang. Formal specification and analysis of Zeroconf using Uppaal. *ACM Trans. Embedded Comput. Syst.*, 10(3):34, 2011.

[BM02] L. Bölöni and D. C. Marinescu. Robust scheduling of metaprograms. *Journal of Scheduling*, 5(5):395–412, 2002.

[BT01] J.-Y. L. Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *LNCS*. Springer, 2001.

[BVBG+10] T. Basten, E. Van Benthum, M. Geilen, M. Hendriks, F. Houben, G. Igna, F. Reckers, S. De Smet, L. Somers, E. Teeselink, N. Trčka, F. Vaandrager, J. Verriet, M. Voorhoeve, and Y. Yang. Model-Driven Design-Space Exploration for Embedded Systems: The Octopus Toolset. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 6415 of *LNCS*, pages 90–105. Springer, 2010.

[BY03] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer, 2003.

[CDF+05] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In *International Conference on Concurrency Theory (CONCUR)*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.

[CGP01] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[CJL+09] F. Cassez, J. J. Jessen, K. G. Larsen, J.-F. Raskin, and P.-A. Reynier. Automatic Synthesis of Robust and Optimal Controllers - An Industrial Case Study. In *HSCC*, volume 5469 of *LNCS*, pages 90–104. Springer, 2009.

[CKM01] S. Christensen, L. M. Kristensen, and T. Mailund. A Sweep-Line Method for State Space Exploration. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 450–464. Springer, 2001.

[CKT03] S. Chakraborty, S. Künzli, and L. Thiele. A General Framework for Analysing System Properties in Platform-Based Embedded System Designs. In *Design, Automation and Test in Europe (DATE)*, pages 10190–10195. IEEE Computer Society, 2003.

[CL00] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *International Conference on Concurrency Theory (CONCUR)*, volume 1877 of *LNCS*, pages 138–152. Springer, 2000.

[CPR08] A. Cimatti, L. Palopoli, and Y. Ramadian. Symbolic computation of schedulability regions using parametric timed automata. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 80–89. IEEE Computer Society, 2008.

[DB00] A. Davenport and J. Beck. A Survey of Techniques for Scheduling with Uncertainty. `www.eil.utoronto.ca/profiles/chris/gz/uncertainty-survey.ps`, 2000. Unpublished manuscript.

[DDL+12] A. David, D. Du, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, and S. Sedwards. Statistical model checking for stochastic hybrid systems. In *International Workshop on Hybrid Systems and Biology*, volume 92 of *EPTCS*, pages 122–136. Open Publishing Association, 2012.

[DK95] R. L. Daniels and P. Kouvelis. Robust scheduling to hedge against processing time uncertainty in single-stage production. *Management Science*, 41:363–376, February 1995.

[DL08] L. G. Ding and L. Liu. Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets. In *International Conference on Applications and Theory of Petri Nets (PETRI NETS)*, volume 5062 of *LNCS*, pages 132–151. Springer, 2008.

[DLL62] M. Davis, G. Logemann, and D. W. Loveland. A machine program for theorem-proving. *Communications of the ACM (CACM)*, 5(7):394–397, 1962.

[DLL+11] A. David, K. G. Larsen, A. Legay, M. Mikučionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *Formal Modeling and Analysis of Timed Systems*, volume 6919 of *LNCS*, pages 80–96. Springer, 2011.

[dMB08] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS, pages 337–340. Springer, 2008.

[dMB09] L. M. de Moura and N. Bjørner. Satisfiability Modulo Theories: An Appetizer. In *Brazilian Symposium on Formal Methods*, volume 5902 of *LNCS*, pages 23–36. Springer, 2009.

[EKK08] K. L. Espensen, M. K. Kjeldsen, and L. M. Kristensen. Modelling and Initial Validation of the DYMO Routing Protocol for Mobile Ad-Hoc Networks. In *International Conference on Applications and Theory of Petri Nets (PETRI NETS)*, volume 5062 of *LNCS*, pages 152–170. Springer, 2008.

[FJT07] J. M. Fernandes, J. B. Jørgensen, and S. Tjell. Requirements Engineering for Reactive Systems: Coloured Petri Nets for an Elevator Controller. In *Asia-Pacific Software Engineering Conference (APSEC)*, pages 294–301. IEEE Computer Society, 2007.

[FKPY07] E. Fersman, P. Krcál, P. Pettersson, and W. Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, 2007.

[FLT09] U. Fahrenberg, K. G. Larsen, and C. R. Thrane. Verification, Performance Analysis and Controller Synthesis for Real-Time Systems. In *IPM International Conference on Fundamentals of Software Engineering (FSEN)*, volume 5961 of *LNCS*, pages 34–61. Springer, 2009.

[FM06] Z. Fu and S. Malik. On Solving the Partial MAX-SAT Problem. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.

[FP12] B. Finkbeiner and H.-J. Peter. Template-Based Controller Synthesis for Timed Systems. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *LNCS*, pages 392–406. Springer, 2012.

[FVVC06] O. Florescu, J. Voeten, M. Verhoef, and H. Corporaal. Reusing Real-Time Systems Design Experience. In *Forum on Specification and Design Languages (FDL)*, pages 375–381. ECSI, 2006.

[GdHN+08] P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating Software Testing Using Program Analysis. *IEEE Software*, 25(5):30–37, 2008.

[GGS+06] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 25–36. IEEE Computer Society, 2006.

[GKM+08] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F. L. Wurden. Model-Based Quality Assurance of Windows Protocol Documentation. In *International Conference on Software Testing, Verification, and Validation (ICST)*, pages 502–506. IEEE Computer Society, 2008.

[GPR98] M. L. Ginsberg, A. J. Parkes, and A. Roy. Supermodels and Robustness. In *National Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 334–339. AAAI Press / The MIT Press, 1998.

[GSB+07] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. D. Theelen. Latency Minimization for Synchronous Data Flow Graphs. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD)*, pages 189–196. IEEE, 2007.

[HBL+03] M. Hendriks, G. Behrmann, K. G. Larsen, P. Niebert, and F. W. Vaandrager. Adding Symmetry Reduction to Uppaal. In *International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 2791 of *LNCS*, pages 46–59. Springer, 2003.

[Her99] J. W. Herrmann. A Genetic Algorithm for Minimax Optimization Problems. In *Congress on Evolutionary Computation*, volume 2, pages 1099–1103. IEEE Press, 1999.

[HGGM01] M. G. Harbour, J. G. García, J. P. Gutiérrez, and J. D. Moyano. MAST: Modeling and Analysis Suite for Real Time Applications.

In *Euromicro Conference on Real-Time Systems (ECRTS)*, pages 125–134. IEEE Computer Society, 2001.

[HHJ+05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis - the SymTA/S approach. *IEE Proceedings Computers and Digital Techniques*, 152(2):148–166, 2005.

[HHW04] E. Hebrard, B. Hnich, and T. Walsh. Super Solutions in Constraint Programming. In *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR)*, volume 3011 of *LNCS*, pages 157–172. Springer, 2004.

[HHWT97] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A Model Checker for Hybrid Systems. In *International Conference on Computer Aided Verification (CAV)*, volume 1254 of *LNCS*, pages 460–463. Springer, 1997.

[HIV12] F. Houben, G. Igna, and F. W. Vaandrager. Modeling Task Systems Using Parameterized Partial Orders. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 317–327. IEEE, 2012.

[HJMS03] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. In *International SPIN Workshop on Model Checking of Software (SPIN)*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[HJRE04] A. Hamann, M. Jersak, K. Richter, and R. Ernst. Design Space Exploration and System Optimization with SymTA/S– Symbolic Timing Analysis for Systems. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 469–478. IEEE Computer Society, 2004.

[HP07] J. Håkansson and P. Pettersson. Partial order reduction for verification of real-time components. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 4763 of *LNCS*, pages 211–226. Springer, 2007.

[HS07] T. A. Henzinger and J. Sifakis. The Discipline of Embedded Systems Design. *IEEE Computer*, 40(10):32–40, 2007.

[HSA] HSA (Heterogeneous System Architecture) Foundation. `http://hsafoundation.com`.

[HV06] M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *International Parallel and*

Distributed Processing Symposium (IPDPS), pages 179–179. IEEE, 2006.

[HvdNV03] M. Hendriks, B. van den Nieuwelaar, and F. Vaandrager. Recognizing finite repetitive scheduling patterns in manufacturing systems. In *Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA)*, pages 291–319. The University of Nottingham, 2003.

[HvdNV06] M. Hendriks, B. van den Nieuwelaar, and F. W. Vaandrager. Model checker aided design of a controller for a wafer scanner. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):633–647, 2006.

[IKY+08] G. Igna, V. Kannan, Y. Yang, T. Basten, M. Geilen, F. W. Vaandrager, M. Voorhoeve, S. de Smet, and L. J. Somers. Formal Modeling and Scheduling of Datapaths of Digital Document Printers. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 5215 of *LNCS*, pages 170–187. Springer, 2008.

[IV10] G. Igna and F. W. Vaandrager. Verification of Printer Datapaths Using Timed Automata. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 6416 of *LNCS*, pages 412–423. Springer, 2010.

[JCTX06] J. B. Jørgensen, S. Christensen, A.-P. Tuovinen, and J. Xu. Tool Support for Estimating the Memory Usage of Mobile Phone Software. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):531–545, 2006.

[Jen92] K. Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*. EATCS Series. Springer, 1992.

[Jen01] M. T. Jensen. *Robust and Flexible Scheduling with Evolutionary Computation*. PhD thesis, BRICS, Department of Computer Science, University of Aarhus, Denmark, 2001.

[JK09] K. Jensen and L. M. Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.

[JKW07] K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(3-4):213–254, 2007.

[JRLD07] J. Jessen, J. Rasmussen, K. Larsen, and A. David. Guided Controller Synthesis for Climate Controller Using Uppaal Tiga. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 4763 of *LNCS*, pages 227–240. Springer, 2007.

[Kas05] A. Kasperski. Minimizing maximal regret in the single machine sequencing problem with maximum lateness criterion. *Operations Research Letters*, 33(4):431 – 436, 2005.

[KDV00] P. Kouvelis, R. L. Daniels, and G. Vairaktarakis. Robust scheduling of a two-machine flow shop with uncertain processing times. *IIE Transactions*, 32:421–432, 2000.

[KDVvdW97] B. Kienhuis, E. F. Deprettere, K. A. Vissers, and P. van der Wolf. An Approach for Quantitative Analysis of Application-Specific Dataflow Architectures. In *International Conference on Application-Specific Systems, Architectures, and Processors (ASAP)*, pages 338–349. IEEE Computer Society, 1997.

[KSY07] P. Krcál, M. Stigge, and W. Yi. Multi-processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times. In *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 4763 of *LNCS*, pages 274–289. Springer, 2007.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LM87] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, 1987.

[LPT09] K. Lampka, S. Perathoner, and L. Thiele. Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems. In *International Conference on Embedded Software (EMSOFT)*, pages 107–116. ACM, 2009.

[LPY97] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1–2):134–152, 1997.

[Maz88] A. W. Mazurkiewicz. Compositional Semantics of Pure Place/ Transition Systems. In *European Workshop on Applications and Theory of Petri Nets*, volume 340 of *LNCS*, pages 307–330. Springer, 1988.

[MBS89] K. McKay, J. Buzacott, and F. Safayeni. The scheduler's knowledge of uncertainty: The missing link. In *Knowledge Based Production Management Systems*, pages 171–189. Elsevier Science, 1989.

[McM92] K. L. McMillan. Using Unfoldings to Avoid the State Explosion Problem in the Verification of Asynchronous Circuits. In *International Conference on Computer Aided Verification (CAV)*, volume 663 of *LNCS*, pages 164–177. Springer, 1992.

[MFC01] A. K. Mok, A. X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS)*, pages 75–84. IEEE Computer Society, 2001.

[MLR+10] M. Mikucionis, K. G. Larsen, J. I. Rasmussen, B. Nielsen, A. Skou, S. U. Palm, J. S. Pedersen, and P. Hougaard. Schedulability Analysis Using Uppaal: Herschel-Planck Case Study. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, volume 6416 of *LNCS*, pages 175–190. Springer, 2010.

[MPS95] O. Maler, A. Pnueli, and J. Sifakis. On the Synthesis of Discrete Controllers for Timed Systems. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 900 of *LNCS*, pages 229–242. Springer, 1995.

[MSB98] K. McKay, F. Safayeni, and J. Buzacott. Job-shop scheduling theory: What is relevant? *Interfaces*, 18(4):84–90, 1998.

[MW99] K. McKay and V. Wiers. Unifying the Theory and Practice of Production Scheduling. *Journal of Manufacturing Systems*, 18(4):241–255, 1999.

[NMFR03] N. v. d. Nieuwelaar, J. v. d. Mortel-Fronczak, and J. Rooda. Design of supervisory machine control. `http://www.nt.ntnu.no/users/skoge/prost/proceedings/ecc03/pdfs/469.pdf`, 2003.

[NO79] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NWY99] C. Norström, A. Wall, and W. Yi. Timed Automata as Task Models for Event-Driven Systems. In *International Workshop on Real-Time Computing and Applications Symposium (RTCSA)*, pages 182–189. IEEE Computer Society, 1999.

[Oct11] Octopus Toolset homepage. `http://dse.esi.nl`, 2011.

[OFCF11] A. Orlandini, A. Finzi, A. Cesta, and S. Fratini. TGA-Based Controllers for Flexible Plan Execution. In *Annual German Conference on AI (KI)*, volume 7006 of *LNCS*, pages 233–245. Springer, 2011.

[Pel98] D. Peled. Ten Years of Partial Order Reduction. In *International Conference on Computer Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.

[PLT11] S. Perathoner, K. Lampka, and L. Thiele. Composing heterogeneous components for system-wide performance analysis. In *Design, Automation and Test in Europe (DATE)*, pages 842–847. IEEE, 2011.

[Pol05] N. Policella. *Scheduling with uncertainty: A proactive approach using Partial Order Schedules.* PhD thesis, University of Rome "La Sapienza", 2005.

[Pra86] V. Pratt. Modeling concurrency with partial orders. *International Journal of Parallel Programming*, 15:33–71, 1986.

[PSCO04] N. Policella, S. F. Smith, A. Cesta, and A. Oddi. Generating Robust Schedules through Temporal Flexibility. In *International Conference on Automated Planning and Scheduling (ICAPS)*, pages 209–218. AAAI, 2004.

[PV97] P. Putten and J. Voeten. *Specification of Reactive Hardware/Software Systems: The Method Software/hardware Engineering (SHE).* PhD thesis, 1997.

[PWT+07] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *International Conference on Embedded Software (EMSOFT)*, pages 193–202. ACM, 2007.

[RJE03] K. Richter, M. Jersak, and R. Ernst. A Formal Approach to MpSoC Performance Verification. *IEEE Computer*, 36(4):60–67, 2003.

[RT06] S. Ranise and C. Tinelli. The SMT-LIB Standard: Version 1.2. `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r12.09.09.pdf`, 2006.

[SBYT12] L. Schor, I. Bacivarov, H. Yang, and L. Thiele. Worst-Case Temperature Guarantees for Real-Time Applications on Multi-core Systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 87–96. IEEE Computer Society, 2012.

[Sch04] K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.

[SG09] I. Sabuncuoglu and S. Goren. Hedging production schedules against uncertainty in manufacturing environment with a review of robustness and stability research. *International Journal of Computer Integrated Manufacturing*, 22(2):138–157, 2009.

[SGB06] S. Stuijk, M. Geilen, and T. Basten. SDF^3: SDF For Free. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 276–278. IEEE Computer Society, 2006.

[SGB08] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[SHE] SHESim tool. `http://www.ics.ele.tue.nl/m̃geilen/shesim`.

[SJD06] Z. Shi, E. Jeannot, and J. J. Dongarra. Robust task scheduling in non-deterministic heterogeneous computing systems. In *International Conference on Cluster Computing (CLUSTER)*, pages 1–10. IEEE, 2006.

[Sol11] V. M. Solà. *Robustness on resource allocation problems*. PhD thesis, Universitat de Girona, 2011.

[SRIE08] S. Schliecker, J. Rox, M. Ivers, and R. Ernst. Providing accurate event models for the analysis of heterogeneous multiprocessor systems. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 185–190. ACM, 2008.

[SRL+11] A. Simalatsar, Y. Ramadian, K. Lampka, S. Perathoner, R. Passerone, and L. Thiele. Enabling parametric feasibility analysis in real-time calculus driven performance evaluation. In R. K. Gupta and V. J. Mooney, editors, *Proceedings of the 14th International Conference on Compilers, Architecture, and Synthesis for Embedded Systems, CASES 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 155–164. ACM, 2011.

[STSB10] N. Stoimenov, L. Thiele, L. Santinelli, and G. C. Buttazzo. Resource adaptations with servers for hard real-time systems. In *International Conference on Embedded Software (EMSOFT)*, pages 269–278. ACM, 2010.

[Sym] SymTA/S industrial case studies. `http://www.symtavision.com/success.html`.

[TBHH07] L. Thiele, I. Bacivarov, W. Haid, and K. Huang. Mapping Applications to Tiled Multiprocessor Embedded Systems. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 29–40. IEEE Computer Society, 2007.

[TCN00] L. Thiele, S. Chakraborty, and M. Naedele. Real-time Calculus for Scheduling Hard Real-Time Systems. In *International Symposium on Circuits and Systems (ISCAS)*, volume 4, pages 101–104. IEEE, 2000.

[TCWCS92] E. Teruel, P. Chrzastowski-Wachtel, J. M. Colom, and M. Silva. On Weighted T-Systems. In *Application and Theory of Petri Nets*, volume 616 of *LNCS*, pages 348–367. Springer, 1992.

[TFG+07] B. D. Theelen, O. Florescu, M. Geilen, J. Huang, P. H. A. van der Putten, and J. Voeten. Software/Hardware Engineering with the Parallel Object-Oriented Specification Language. In *International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 139–148. IEEE, 2007.

[THB+11] N. Trcka, M. Hendriks, T. Basten, M. Geilen, and L. J. Somers. Integrated model-driven design-space exploration for embedded systems. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (ICSAMOS)*, pages 339–346. IEEE, 2011.

[Thi07] L. Thiele. Performance analysis of distributed embedded systems. In *International Conference on Embedded Software (EMSOFT)*, page 10. ACM, 2007.

[TVB11] N. Trčka, M. Voorhoeve, and T. Basten. Parameterized Partial Orders for Modeling Embedded System Use Cases: Formal Definition and Translation to Coloured Petri Nets. In *International Conference on Application of Concurrency to System Design (ACSD)*, pages 13–18. IEEE, 2011.

[TVK03] B. D. Theelen, J. Voeten, and R. D. J. Kramer. Performance modelling of a network processor using POOSL. *Computer Networks*, 41(5):667–684, 2003.

[Uppa] Uppaal PORT homepage. `http://www.uppaal.org/port`.

[Uppb] Uppaal Tiga homepage. `http://people.cs.aau.dk/~adavid/tiga`.

[vdAvH02] W. M. P. van der Aalst and K. M. van Hee. *Workflow Management: Models, Methods, and Systems.* MIT Press, 2002.

[vGP09] R. J. van Glabbeek and G. D. Plotkin. Configuration structures, event structures and Petri nets. *Theoretical Computer Science*, 410(41):4111–4159, 2009.

[VHL03] G. E. Vieira, J. W. Herrmann, and E. Lin. Rescheduling manufacturing systems: A framework of strategies, policies, and methods. *Journal of Scheduling*, 6:39–62, 2003.

[Win89] G. Winskel. An introduction to event structures. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 364–397. Springer, 1989.

[WMY01] C. Wong, P. Marchal, and P. Yang. Task concurrency management methodology to schedule the MPEG4 IM1 player on a highly parallel processor platform. In *International Symposium on Hardware/Software Codesign (CODES)*, pages 170–177. ACM, 2001.

[WWP09] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM (CACM)*, 52(4):65–76, 2009.

[YGB+10] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal. Automated bottleneck-driven design-space exploration of media processing systems. In *Design, Automation and Test in Europe (DATE)*, pages 1041–1046. IEEE, 2010.

[Z3] Z3 homepage. `http://research.microsoft.com/en-us/um/redmond/projects/z3`.

Summary

Computer systems are part of our daily life being included in various systems such as: home appliances, aircrafts, intelligent highway systems, and multimedia communication systems. As the complexity of today's computer-based systems increases, the design methods that guarantee their correctness also need to be updated. Recently, we observe that industry tends to use formal verification tools more often in the design phase. They benefit, among others, from finding errors early. Formal verification is the process of proving the correctness of a system, represented as an abstract mathematical model, against a given property formally described with mathematical rigor. Examples of formal verification techniques are theorem proving, simulation, testing, and model checking.

The aim of this thesis is to help in the design of computer-based systems by using formal verification techniques, mainly model checking. An example of such a system is an Océ datapath, which we extensively analyze in this thesis. A datapath encompasses the complete trajectory of image data from source (e. g. a scanner) to target (e. g. a printer). Model checking is an automated verification technique that, given a model of finite-state system and a formal property, systematically checks whether the property holds in the model or not. The model checker used in this study is Uppaal. The language of Uppaal is based on timed automata, i.e. finite-state machines extended with real-valued variables called clocks, and extends them with additional features such as bounded integer variables, synchronization channels and user-defined functions.

In Chapter 2, we represent a datapath using three modeling approaches: timed automata, colored Petri nets and synchronous data flow graphs. The three models are configured to encode an Océ-specific problem where a set of concurrent jobs (i. e. applications) have to be scheduled on an architecture with limited resources. Uppaal, CPN Tools and SDF3 are used to derive schedules.

Chapter 3 presents an analysis of two concurrent jobs: one continuously occupying a datapath and the other with uncertain arrival times. This scenario is represented as a two-player timed game, and with Uppaal Tiga we seek acceptable trade-offs between the throughput of the continuous job and the latency of the other job.

In Chapter 4, we refine the timed automata model of the Océ datapath by adding scheduling rules specific to the Océ controller. This model is further ana-

lyzed with Uppaal for finding the worst case latency of a job that has uncertain arrival times in a setting where the system is processing in parallel an infinite stream of another competing job.

In Chapter 5, we define real-time task systems (RTTSs) as a general model for real-time systems. In this thesis, we assume that a job contains a finite set of tasks. The set of tasks is represented in an RTTS as a parametrized partial order (PPO), which has been introduced before to compactly represent large task graphs with repetitive behavior. Further, we present a subclass of PPOs that can be efficiently translated to timed automata. Lastly, we report on a series of experiments which demonstrates that models generated from the RTTS representation are more tractable than the handcrafted models of Chapter 2.

In Chapter 6, we scrutinize the validity of the schedules presented in the other chapters and obtained with Uppaal. The duration of some tasks in these schedules is over-approximated, which might make the schedules invalid if one assumes that idle components do not wait (so-called "non-lazy scheduling") when executing tasks with the exact duration. For the schedules proven invalid, we investigate their robustness by slightly varying some task durations. Both validity and robustness analyses are performed with Z3, a satisfiability modulo theory solver. Since tasks run on physical machines, their duration cannot always be precisely known in advance, but is estimated. Therefore, it is important to study the robustness of these schedules when some preconditions are modified. The work in this chapter can be applied in such a situation.

Even though it required some expertise to model an Océ datapath efficiently, Uppaal has been shown valuable for the comparison of various design options.

Samenvatting

Computersystemen maken deel uit van ons dagelijks leven, omdat ze in verschillende systemen zijn ingebed: bv. een huishoudelijk apparaat, een vliegtuig, een intelligente snelweg of een multimediaal communicatiesysteem. Omdat de complexiteit van dergelijke systemen met ingebedde computer tot heden toeneemt, moeten ook de ontwerpmethoden die hun correctheid waarborgen op de nieuwste stand gebracht worden. In de laatste tijd, zo stellen wij vast, neigt de industrie meer ernaar gereedschappen voor formele verificatie tijdens de ontwerpfase te gebruiken; dat heeft o.a. het voordeel dat fouten vroeg gevonden worden. Formele verificatie is het proces waarbij de correctheid van een systeem, beschreven door een abstract wiskundig model, ten opzichte van een eveneens formeel opgeschreven (gewenste) eigenschap met wiskundige strengheid bewezen wordt. Voorbeelden van formele verificatie zijn automatisch bewijzen, simulatie, testen en model checking.

Het doel van dit proefschrift is het ontwerp van systemen met ingebedde computer ondersteunen door gebruik te maken van formele verificatietechnieken, vooral van model checking. In dit proefschrift analyseren we een voorbeeld van een dergelijk systeem uitvoerig: een gegevenspad, dat is het complete pad dat beeldgegevens van een bron (bv. een scanner) naar een doel (bv. een printer) afleggen, van Océ. Model checking is een geautomatiseerde bewijsmethode die van een model van een systeem met eindig veel toestanden systematisch onderzoekt of een gegeven formele eigenschap in het model geldt of niet. Wij gebruiken een model checker namens Uppaal. Deze model checker werkt met tijdsautomaten; dat zijn automaten met eindig veel toestanden, uitgebreid met variabelen, zogenaamde klokken, die reële waarden kunnen aannemen. De modelleertaal van Uppaal breidt tijdsautomaten verder uit met een aantal extra mogelijkheden: variabelen die een beperkt aantal gehele getallen kunnen aannemen, synchronisatiekanalen en door de gebruiker gedefiniëerde functies.

In hoofdstuk 2 modelleren we een gegevenspad op drie verschillende manieren: met tijdsautomaten, gekleurde Petrinetten en synchrone datastroomgrafen. De drie modellen zijn zó vormgegeven dat ze een Océ-specifiek probleem oplossen: een verzameling van tegelijk lopende jobs (toepassingen) moet op een architectuur met beperkte mogelijkheden ingeroosterd worden. We produceren roosters met behulp van Uppaal, CPN Tools en SDF3.

Hoofdstuk 3 stelt een analyse van twee tegelijk lopende toepassingen voor:

de ene taak bezet een bepaald gegevenspad voortdurend en de andere heeft een onbekende starttijd. Dit scenario wordt gemodelleerd als een tijdsspel met twee spelers. Met behulp van Uppaal Tiga zoeken we strategieën die een aanneembare afweging tussen de capaciteit van de voortdurende toepassing en de langstmogelijke vertraging van de andere toepassing veiligstellen.

In hoofdstuk 4 breiden we het tijdsautomaten-model van het Océ-gegevenspad uit: we voegen regels voor de roostering toe, die specifiek voor de Océ-besturingseenheid gelden. We analyseren het daarna met Uppaal om de langstmogelijke vertraging van een toepassing met een onbekende starttijd vast te stellen, terwijl het systeem ook een eindeloze stroom van andere, concurrerende toepassingen verwerkt.

In hoofdstuk 5 definiëren we realtime-taaksystemen (RTTS), een algemeen model om ingebedde systemen met echte tijd te beschrijven. In ons werk nemen we aan dat een toepassing een eindig aantal taken omvat. Een verzameling van taken wordt in RTTS voorgesteld als een geparametriseerde partiële ordening (PPO), een structuur die elders is ingevoerd om grote taakgrafen met herhalingen in het gedrag weer te geven. Daarna beschrijven we een subklasse van PPOs die efficiënt naar tijdsautomaten vertaald kunnen worden. Tot slot leggen we verslag af van een reeks experimenten, die aantoont dat de van RTTS afgeleide modellen beter handelbaar zijn dan de handgemaakte modellen uit hoofdstuk 2.

In hoofdstuk 6 nemen we de geldigheid van de roosters die in de andere hoofdstukken met Uppaal aangemaakt zijn onder de loep. De duur van sommige taken in deze roosters is een (langere) schatting; daardoor zouden sommige roosters ongeldig kunnen worden als we de exacte duur gebruiken en aannemen dat de onderdelen niet wachten, maar direct de volgende taak beginnen zodra ze vrijkomen. We onderzoeken de robuustheid van deze ongeldige roosters door de duur van enkele taken iets te wijzigen. De geldigheid- en robuustheid-analyses worden beide uitgevoerd met Z3, een SMT-solver, dat is een gereedschap dat geldigheid onder een theorie bewijst. Bij taken die op fysieke machines uitgevoerd worden is het soms niet mogelijk van te voren de exacte tijdsduur aan te geven, maar alleen een schatting. Daarom is het belangrijk de robuustheid van dergelijke roosters te onderzoeken, als sommige voorwaarden wijzigen. De methode van dit hoofdstuk kan ook hiervoor gebruikt worden.

Alhoewel enige expertise nodig was om het Océ-gegevenspad efficiënt te modelleren, kunnen we vaststellen dat Uppaal zijn waarde bij het vergelijken van de verschillende ontwerpkeuzes heeft bewezen.

vertaling: David N. Jansen.

Curriculum Vitae

Born on 15.02.1983 in Arad, Romania.

2001–2006 Dipl.-Ing., Faculty of Automation and Computing, Politehnica University of Timişoara, Romania.

2006–2007 Test Engineer, Océ Timişoara, Romania.

2007–2007 Junior Researcher, e-Austria Institute, Timişoara, Romania.

2007–2011 PhD student, Model Based System Development group, Radboud University of Nijmegen, The Netherlands.

Acknowledgements

I am indebted to many people for making the time working on my PhD study a wonderful experience.

First, I would like to express my sincere gratitude to my promoter, Frits Vaandrager for his guidance, research suggestions and support during my PhD study. Frits, I really appreciate your patience and mentoring skills.

I would like to thank the members of my manuscript committee: Erika Ábráham, Twan Basten, Alexandre David, Jozef Hooman and Marius Minea for taking the time to review and comment on my thesis. To Marius I am also grateful for his sustained guidance and support he has given me over the years.

I am also grateful to my colleagues in Octopus Project. Our discussions on Tuesdays were always useful for my research. Thank you Frans, Twan, Jacques, Lou, and Sebastian for your feedback and help to advance my understanding of the Océ systems. Martijn and Nikola gave me a lot of inspiration from their experience. Yang and Venkatesh were always fun to talk on different matters, such as life as PhD students. I would like to thank Marc for how he always showed positive feedback for my presentations, may his soul rest in peace. I am also grateful to Sander for driving me to Venlo.

I would like to thank everyone at the MBSD group for being social and friendly and for creating a nice working environment. I would like to thank my office mates: Wenyun, Fides, Fiona, Ilona, Agnes, Denis, Michele, and Jasper for our discussions, gatherings and 'gezellig' atmosphere. I would like to thank Faranak and Ali for sharing with me a lot from their rich culture and being always so kind to me. Thank you David for our discussions on topics related to work but also life-related, and for your help with the translation of my summary in Dutch. I thank Jan, Lars and Julien for the teaching skills I have learned from them. I thank Harco, John, Bernard (Security group) for their help in technical problems. Thank you Freek for the discussions we have had on our research. I am grateful to Irma and Ingrid for all their help with documents and practical duties. Many thanks Irma for our Dutch lessons - because of you now I can say that "Ik kan een beetje Nederlands praten". Marina, often after talking to you I felt myself a lot more positive, thank you for this. Thank you Alexandra (Foundations group) for being such a friendly room-mate during our QAPL summer school in Bertinoro; I was happy to see you again in Nijmegen.

Many thanks to Kim Larsen and Jacob Illum Rasmussen for their help and guidance during my research visit in their group.

I would like to thank Audrey, Rita, Andreea, Adriana, Botond and Lucian for their unconditional help and support. Thank you Olha for many trips and cultural events where we went together. Special thanks to Betty for always finding her so close despite differences in distance and time zones. Many thanks to my Familia (Luminita, Cezary, Pavol and Patrjcya). I really miss our philosophical discussions and social activities from the times when we lived together. I would like to thank the 'lost in Nijmegen' group: Gleb, Alfi, Andrew, Maria, Silvia, Irene and Peer for the wonderful activities that we had together, among which a 4-daagse training. Next year I'll try participate ;) Thank you Betty, Adina, Claudiu and Tudor for visiting me in Nijmegen.

Further, I would like to thank my family and other friends in Romanian.

Doresc să mulțumesc Familiilor Bubberman și Muntean-Engles pentru căldura și ajutorul pe care mi le-au oferit. De asemenea, Alina și doamnele Marieta și Nicoleta au fost intotdeauna alături de bine cu o vorbă bună si o incurajare, vă mulțumesc. O deosebită plăcere mi-au făcut și întalnirile cu familiile Jdira, Dorobanțu și Banea. Mulțumiri, de asemenea, și Parinților Ioan și Costel și familiilor lor pentru exemplu și îndrumare.

Călătoriile mele acasă nu ar fi fost mai plăcute, daca dragii mei prieteni Mihaela, Dan, Mihaela, David si Crenguța nu m-ar fi întampinat cu atata prietenie. Mulțumesc de asemenea Emiliei, matușilor Anișoara, Lenuța, și Ileana și unchiului Ionică pentru ca au fost intotdeauna alaturi de familia noastra. Mulțumesc surorii mele Liliana, cumantului Nicușor și nepotului Ionuț pentru căldura, ajutorul și încurajările acordate. Mulțumiri mamei și tatalui meu pentru dragostea, înțelegerea, sacrificiul și sprijinul financiar pe care mi le-au oferit de-a lungul anilor. Cu tata, din pacate, nu o sa ne mai vedem pentru o vreme...

Titles in the IPA Dissertation Series since 2007

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation.* Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains.* Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use.* Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems.* Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML.* Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery.* Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding.* Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks.* Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data.* Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy.* Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS.* Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design.* Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems.* Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms.* Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications.* Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation.* Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems.* Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification.* Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development.* Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation.* Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean.* Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques.* Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time.* Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation.* Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML.* Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange.* Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web.* Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory.* Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems.* Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models.* Faculty of Electrical Engineer-

ing, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis.* Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks.* Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement.* Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers.* Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics.* Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics.* Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata.* Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software.* Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra.* Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications.* Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services.* Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols.* Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code*

Generation with Templates. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement.* Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components.* Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems.* Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors.* Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory.* Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model coevolution.* Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes.* Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains.* Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification.* Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations.* Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice.* Faculty of Science,

Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars.* Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors.* Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns.* Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems.* Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures.* Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited.* Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference.* Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization.* Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation.* Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10

L.J.P. Engelen. *From Napkin Sketches to Reliable Software.* Faculty of Mathematics and Computer Science, TU/e. 2012-11

F.P.M. Stappers. *Bridging Formal Models – An Engineering Perspective.* Faculty of Mathematics and Computer Science, TU/e. 2012-12

W. Heijstek. *Software Architecture Design in Global and Model-Centric Software Development.* Faculty of

Mathematics and Natural Sciences, UL. 2012-13

C. Kop. *Higher Order Termination*. Faculty of Sciences, Department of Computer Science, VUA. 2012-14

A. Osaiweran. *Formal Development of Control Software in the Medical Systems Domain*. Faculty of Mathematics and Computer Science, TU/e. 2012-15

W. Kuijper. *Compositional Synthesis of Safety Controllers*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-16

H. Beohar. *Refinement of Communication and States in Models of Embedded Systems*. Faculty of Mathematics and Computer Science, TU/e. 2013-01

G. Igna. *Performance Analysis of Real-Time Task Systems using Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2013-02